(2)

Debugging and Analysis of Large-Scale Parallel Programs

John M. Mellor-Crummey

Technical Report 312
September 1989

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Debugging and Analysis of Large-Scale Parallel Programs

by

John M. Mellor-Crummey

Submitted in Partial Fulfillment

of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

Supervised by Thomas J. LeBlanc

Department of Computer Science

University of Rochester

Rochester, New York

September 1989

Curriculum Vitae

John Michael Mellor-Crummey was born John Michael Crummey on August 4, 1962, in Jersey City, NJ. He grew up in Livingston, NJ, attending public schools until 1980. During the 1979–1980 academic year, he participated in the Science Honors Program at Columbia University where he studied Particle Physics and Astronomy, until a transit strike cut off his access to New York City. In the fall of 1980, he entered Princeton University to study Chemical Engineering. After a semester of Chemical Engineering (which turned out to have little in common with Chemistry, which he enjoyed), he switched his major to Electrical Engineering and Computer Science. In the spring of 1983, he was elected to Tau Beta Pi, the national engineering honor society. In June, 1984, he earned a B.S.E. in Electrical Engineering and Computer Science with high honors from Princeton, was elected to Phi Beta Kappa, and was awarded an honorable mention in the National Science Foundation graduate fellowship competition. In the fall of 1984, he entered the graduate program in Computer Science at the University of Rochester as a Sproull Fellow. In 1985, he was awarded another honorable mention in the National Science Foundation graduate fellowship competition, but alas, again no money. During his first two years in Rochester, he served as a teaching assistant for Introduction to Artificial Intelligence, a research assistant for John Ellis performing VLSI layout of a ray-casting chip (for rendering scenes described using constructive solid geometry), and a teaching assistant for Introduction to Automata Theory. In May, 1986, he was awarded a M.S. in Computer Science. From the spring of 1986 until his graduation he served as a research assistant for Prof. Thomas LeBlanc working on debugging and analysis of large-scale parallel programs.

Acknowledgments

First and foremost, I would like to thank Tom LeBlanc, my advisor, for his invaluable contribution to the work described in this dissertation. Tom's pointed criticisms and questions shaped the direction of this research and his ability to examine a problem in the abstract often helped me "see the forest from amidst the trees". Finally, Tom's careful reading of my drafts and his insightful comments substantially improved the clarity of the final version.

Thanks are also due to the other members of the Parallel Program Understanding Tools and Techniques (PPUTTS) group: Rob Fowler, Neil Smithline and Ivan Bella. Rob, a member of my dissertation committee, deserves thanks for his interest and collaboration on the development of the integrated toolkit. Rob was also the primary supervisor of Ivan, who coded the Moviola graphical interface for examining execution traces. Neil deserves thanks for his work "gluing" Moviola to KCL, which made the integrated toolkit prototype a reality.

I would also like to thank Michael Scott and Bruce Arden, the other members of my dissertation committee, for their interest and feedback on my work. Michael's detailed comments on my final draft were greatly appreciated.

Thanks are also due to the systems grad students in the department, especially Lawrence Crowl, Peter Dibble, and Neal Gafter. They were always willing to listen to new results, critique ideas or algorithms, and provide comments on drafts of my work. Neal's insight was responsible for uncovering subtle flaws in early versions of several algorithms I developed.

Last, but certainly not least, I would like to thank my wife Cindy. Without her love, support, and understanding, I would not have made it this far. I thank her for all of the hours that she patiently waited "just one more minute while I finish this up".

This work was supported in part by U.S. Army Engineering Topographic Labs research contract no. DACA76-85-C-0001, NSF research contract no. CCR-8704492, and ONR research contract no. N00014-87-K-0548.

Abstract

One of the most serious problems in the development cycle of large-scale parallel programs is the lack of tools for debugging and performance analysis. Parallel programs are more difficult to analyze than their sequential counterparts for several reasons. First, race conditions in parallel programs can cause non-deterministic behavior, which reduces the effectiveness of traditional cyclic debugging techniques. Second, invasive, interactive analysis can distort a parallel program's execution beyond recognition. Finally, comprehensive analysis of a parallel program's execution requires collection, management, and presentation of an enormous amount of information.

This dissertation addresses the problem of debugging and analysis of large-scale parallel programs executing on shared-memory multiprocessors. It proposes a methodology for top-down analysis of parallel program executions that replaces previous ad-hoc approaches. To support this methodology, a formal model for shared-memory communication among processes in a parallel program is developed. It is shown how synchronization traces based on this abstract model can be used to create *indistinguishable* executions that form the basis for debugging. This result is used to develop a practical technique for tracing parallel program executions on shared-memory parallel processors so that their executions can be repeated deterministically on demand. Next, it is shown how these traces can be augmented with additional information that increases their utility for debugging and performance analysis. The design of an integrated, extensible toolkit based on these traces is proposed. This toolkit uses execution traces to support interactive, graphics-based, top-down analysis of parallel program executions. A prototype implementation of the toolkit is described explaining how it exploits our execution tracing model to facilitate debugging and analysis. Case studies of the behavior of several versions of two parallel programs are presented to demonstrate both the utility of our execution tracing model and the leverage it provides for debugging and performance analysis.

Table of Contents

List of Tables

List of Figures

1 Introduction

The demand for increased computing power has been a driving force behind advances in digital computers. As fabrication technology for high-performance processing elements approaches the limits of device physics, incremental increases in the performance of single-processor computer systems come at great cost. The steepness of the price/performance curve for high-performance single-processor computers has spurred the development of parallel computer architectures that are fashioned as a set of processing elements linked by an interconnection network. High-performance parallel computers cost a fraction of the price of comparably powerful single-processor systems. However, the low price/performance ratio for parallel computers is not without cost: parallel programs are highly complex.

Parallel programming inherits all the traditional difficulties of software development, as well as many that arise out of the introduction of parallelism. The two most daunting difficulties unique to parallel programming are partitioning the computation task into multiple, balanced components and implementing an efficient strategy for coordinating the efforts of these components. These additional demands on programmers make development of parallel software a difficult, error-prone process.

Since parallel software development is so error-prone, tools for debugging and analysis are essential to support the development cycle of large-scale parallel programs. Several issues complicate analysis of parallel program executions. First, they often exhibit nonrepeatable behavior, which makes them difficult to understand and problems difficult to pinpoint. Second, interactive analysis can distort executions beyond recognition. Third, comprehensive analysis requires collection, management, and presentation of an enormous amount of data. Finally, tools and techniques for debugging and analysis of parallel program executions must be integrated into an environment that admits extensive parallelism during computation, but provides a single user-interface during analysis.

This dissertation focuses on the development of techniques for debugging and analyzing large-scale parallel programs. While most of the techniques are general, applicable to analysis of parallel programs in both loosely-coupled and tightly-coupled domains, this work emphasizes the analysis of parallel programs for shared-memory multiprocessors.

Shared-memory multiprocessors provide unique challenges for debugging and analysis. Programs on these machines have the potential to use shared memory as a fine-grained, high-bandwidth, low-latency communication medium. This potential places

rigorous demands on the tools and techniques that can be used. Monitoring tools must be able to record a large number of fine-grain events without distorting a program's execution. Since programs can use shared memory without kernel intervention, they must incorporate their own software instrumentation. Instrumentation must be decentralized, since multiprocessors often lack a single, easily monitored, shared communication medium. Since shared-memory multiprocessors can support a wide variety of communication and synchronization abstractions (including direct use of shared memory, message passing, and remote procedure call), instrumentation must be flexible enough to be customized for different programming models. Finally, analysis tools must be general and extensible to support manipulation of events and abstractions that are appropriate for each different model of programming.

1.1 Parallel Program Development

To use a parallel computer effectively, tasks must be broken down into independent units of computation that can be performed in parallel. The form of an efficient parallel algorithm is intimately coupled to the parallel computer architecture on which it will execute. Two alternative organizations of parallel computers are single-instruction, multiple-data stream (SIMD) and multiple-instruction, multiple-data stream (MIMD) systems. (See [Flynn, 1972] for a detailed comparison of various parallel computer organizations.) In SIMD parallel computers such as the Connection Machine [Hillis, 1985], each processor executes the same sequence of instructions in lock-step on its own data. On MIMD parallel computers (*e.g.*, shared-bus, shared-memory multiprocessors [Sequent, 1984; Encore, 1987]; distributed-memory, shared-memory multiprocessors [Gottlieb *et al.*, 1983; Pfister *et al.*, 1985; BBN Laboratories, 1986]; and distributed-memory, message-passing multiprocessors [Seitz, 1985]), parallel programs consist of multiple asynchronous processes that communicate using some form of message passing or shared memory. No assumption may be made about the relative speed of processes, other than finite progress by each process. Efficient parallel algorithms for SIMD computers have an extremely regular structure and exploit fine-grain parallelism. Algorithms with less regular structure and coarse-grain parallelism are better suited to MIMD computers. In either case, to be efficient, an algorithm must be crafted to use a communication structure that can be imbedded efficiently in the topology of the target parallel computer's interconnection network. We focus here on the difficulties inherent in software development for MIMD parallel computers.

We also focus on imperative programming languages, the current standard for parallel programming. Other language models for parallel programming, including functional, logic, and dataflow languages, have attributes that make them attractive for parallel programming; however, the efficiency of imperative programming languages is currently unmatched. Since the purpose of using parallel computers is speed, programmers are reluctant to use any model of parallel programming that will not provide them with the highest attainable performance. However, parallel programs written in imperative languages generally require explicit communication and synchronization to coordinate the program components.

Several factors make it difficult to use primitives for communication and synchronization correctly. First, using these primitives requires careful attention to detail to ensure that the effects of primitive operations are not lost (*e.g.*, overflow of a bounded-length message queue), ignored (*e.g.*, entering a critical section without using the appropriate access control primitives) or misinterpreted (*e.g.*, a mismatch between actual and formal parameters in a remote procedure call invocation). Second, coordinating multiple components admits the possibility of circular dependencies in communication or synchronization that result in deadlock. Third, there is a potential for race conditions between components in an execution that may cause the program to exhibit non-deterministic behavior (*e.g.*, producing different results for the same set of program inputs).[1] Finally, most environments for parallel programming provide low-level primitives for synchronization and communication, since high-level primitives cannot be tailored efficiently to the problem at hand. However, the availability of low-level primitives is often a liability to the parallel programmer; incorrect implementation of special-purpose abstractions is a common source of error.

Concern for efficiency is an additional burden on the programmer. Parallel programs that use simple synchronization primitives such as barriers may make inefficient use of available processors. Since it is often impossible to evenly balance the size of the tasks assigned to each processor, many processors may sit idle waiting for stragglers to reach a barrier. Similarly, programs that use simple control structures (such as having cooperating processes use a global queue to maintain a list of available work) may also make inefficient use of available parallelism. Using a global work queue can limit the speedup achievable unless the size of the units of work that are allocated from the queue is adjusted according to the number of processors in use. Otherwise, contention for access to the queue can dominate the time processors spend working. Taking full advantage of available parallelism often requires implementation of complicated control strategies and data structures to avoid sequential bottlenecks. Examples of these include using multiple processors to spawn tasks in a binary tree fashion rather than have a single processor sequentially start up child tasks [LeBlanc and Jain, 1987], building distributed implementations of synchronization primitives such as a software combining tree [Yew *et al.*, 1987] (to diffuse hot-spot memory contention), and building concurrent data structures (*e.g.*, a concurrent queue [Mellor-Crummey, 1987]) that permit multiple overlapping operations.

Clearly, parallel programs possess a level of complexity not present in sequential domains. As a result of this complexity, development of parallel programs is difficult and techniques that provide programmers with insight into the execution behavior of their parallel programs (especially regarding synchronization and communication) are needed to help support the development cycle for parallel software.

---

[1]The focus here is on programs that exhibit *true parallelism* or, at the least, appear to exhibit parallelism due to preemptive scheduling of processes. A concurrent program implemented by coroutines running on a single processor without the possibility of preemption can be debugged as if it were a sequential program.

1.2 Analyzing Parallel Program Executions

Debugging and performance analysis are essential to the development of correct and efficient parallel programs. Debugging is the art of locating flaws in a program that have caused erroneous behavior. It is a two-step process that involves observing errors in a program's behavior and diagnosing their causes. Diagnosing errors is a difficult problem since many statements may execute between the time a fault occurs and the time it is detected by a programmer. Tracing an error back to its source usually requires information in addition to that which led to its detection. Performance analysis involves determining whether a program satisfies performance criteria established by its designer, and if not, which parts of the program do not perform adequately. Although performance analysis is important for sequential programs, it is crucial for parallel programs. Even though each process of a parallel program may perform well when tested independently, the program as a whole may perform poorly as a result of interactions between processes during execution.

Unfortunately, many characteristics of parallel domains make it difficult to apply techniques that have proven effective for analysis of sequential program executions. This section briefly reviews the most common techniques used for debugging and performance analysis of sequential programs, examines the assumptions upon which they are based, and shows how these assumptions are violated in parallel domains.

1.2.1 Debugging

Sequential program debugging is a well understood task that draws on tools and techniques developed over many years. One early technique was to record snapshots of the entire program state, often consisting of many pages of hexadecimal digits, for perusal by a programmer. Debugging was a programmer-intensive operation, since there were few tools for analyzing snapshots. Over time this approach was replaced by interactive debuggers, which allow programmers to examine arbitrary details of a program's state during execution. Debugging then became more computation-intensive, since the computer was used to reproduce execution sequences with successively greater detail. As a result, the most common method used today to debug sequential programs is cyclic: a program is executed until an error manifests itself. The programmer then postulates a set of underlying causes for the error, inserts trace statements or additional breakpoints to gather more information, and re-executes the program.[2]

Garcia and Berman [Garcia and Berman, 1985] isolated three important assumptions implicit in cyclic debugging techniques for sequential programs. First, program executions are assumed to be reproducible. The iterative task of narrowing the set of hypotheses about the cause of an error relies on reproducing a program execution for

[2]The problem of optimal placement of tracepoints in a program to discriminate among a set of potential program faults is analogous to the traversal of an optimal binary search tree with the program faults at the leaves. The probability that each potential fault is responsible for the observed error can be used to build an optimal binary search tree. In each pass through the program execution, tracepoints should be added to collect the information necessary to advance a path from the root through another internal node in the tree toward the leaves.

further analysis. Second, programmers are assumed to know where to place tracepoints in their program to differentiate between possible causes for an observed execution error. Third, the presence of tracepoints is assumed not to alter program execution. Complications arise when attempting to use the cyclic debugging method for parallel program executions because they violate the implicit assumptions on which cyclic debugging is based.

The assumption that program behavior is deterministically reproducible is crucial for the success of the cyclic debugging method. Sequential programs are usually deterministic; that is, for a fixed input, each execution of a program always follows the same execution path and produces the same results. However, the behavior of parallel programs (especially erroneous ones) is often non-deterministic; parallel programs do not fully specify all possible execution sequences.[3] Since the behavior of each process in a parallel program execution is a function of its inputs and the values it sees in shared data, the execution behavior of a parallel program in response to a fixed input may be indeterminate, with the results depending on the particular interleaving of processes as they access shared data.

Locking protocols that control access to shared data structures that cannot be updated atomically are necessary to ensure the integrity of the shared data; however, in general, locks are not enough to eliminate non-deterministic behavior. Although locks can ensure that conflicting accesses to shared data do not overlap, they do not usually ensure a particular deterministic order of access (*e.g.*, round robin) to the shared data. Thus, even with locks, access order to shared structures is usually indeterminate. Since an error might occur only in the presence of a particular interleaving of processes, cyclic debugging will be of little use unless the conditions that result in the error are readily reproducible.

The second assumption of the cyclic debugging method is that programmers can determine where to place tracepoints. To do so, a programmer needs to understand the state in which the error occurred. The presence of multiple loci of control makes understanding the failure state of a parallel program very difficult. Typically, understanding a failure state requires inspecting each of the active components of the program. Unless the states of many of the components are identical, it is very difficult to assimilate the global state of a large-scale parallel program in this fashion. Furthermore, the cyclic debugging method assumes that tracepoints will generate useful information that helps isolate the cause of an error. Understanding failure in a parallel program execution requires understanding the dynamic relationship between components executing in parallel. This requires tracepoints in each of the components in a parallel execution. For large-scale parallel programs, the volume of information generated by such tracepoints can be overwhelming.

The third assumption of the cyclic debugging method is that adding tracepoints

[3]It is important to note that incomplete specification of the order of operations in a parallel program is not inherently bad. For example, the particular unit of work allocated during a call to a task generator shared by processes in a parallel program is sensitive to the order in which processes interleave their accesses to the generator; however, sharing a common task generator can provide a substantial performance benefit by balancing the dynamic workload.

to the program does not alter program execution. However, for parallel programs, the overhead of gathering information at tracepoints can bias the set of process interleavings that are likely to occur during an execution by changing the relative cost of code fragments that are executed in parallel.

1.2.2 Performance Analysis

Performance analysis of sequential programs is typically based on examination of execution profiles that show how much time was spent in each block of program code. The most common technique for generating these profiles is based on statistical sampling of the program counter during execution. Sample intervals must be regular; otherwise, execution profiles will not accurately reflect program behavior. Profiles built using coarse-grain sample intervals based on a system real-time clock usually have adequate accuracy. The *gprof* [Graham *et al.*, 1982] utility available on Unix[4] uses this technique to generate instruction-level execution profiles; these instruction profiles are grouped at the procedure level for symbolic presentation. For more accurate statistical profiles, a hardware instruction counter can be used to ensure exactly even sample intervals [Cargill and Locanthi, 1987]. Another technique for generating execution profiles involves associating a count with each basic block in the program. On entry to each basic block, the count is incremented. This technique produces exact execution profiles (basic block entry counts make it possible to determine exactly how many times each instruction has been executed), but results in higher execution overhead.

Since execution profiles provide a measure of how much time was spent in each block of program code, they are useful guides for performance tuning of sequential programs. They provide a measure of how much reducing the execution time of an individual component in the execution (either a procedure, or a basic block) will reduce the total execution time. For parallel programs, such profiles are less useful. The execution time of one component of can be reduced and with no net effect on the overall execution time. A concept from operations research known as the *critical path* [Lockyer, 1964] is useful in explaining this seeming anomaly. A critical path is the longest sequential path among a set of tasks that bounds the overall completion time from below. Reducing the execution time of any component that does not lie along the critical path will have no effect on completion time.

1.3 Statement of Thesis

It is the thesis of this dissertation that although the problems associated with parallel program analysis are different and more complex than those associated with sequential program analysis, the same analysis methodology can be employed for both. Specifically, the sequential program debugging methodology can be extended for parallel program debugging and analysis through the use of dynamic fine-grain characterizations of program executions based on partial orders of accesses to shared data structures. The

[4] Unix is a registered trademark of AT&T

essential characteristics of the traditional methodology that we propose applying to parallel program analysis are as follows.

Analysis is top-down. Program analysis is too complex to require a detailed viewpoint at all times. Abstract views of a program execution are essential to manage complexity, though they may hide relevant details. It must be possible to move from abstract views to concrete details as the focus of interest is narrowed. In particular, for debugging, it must be possible to shift the focus from an entire program to a single process within the program, then to a procedure within the process, and finally to a statement within the procedure.

Analyses are potentially fine-grain. Although many analyses are possible with coarse-grain information, debugging and performance analysis may require access to detailed information. In particular, it must be possible to discover errors related to a single machine-language instruction or a single shared variable. Any reasonable query regarding an execution must be satisfiable.[5] While it may be impractical to store the information needed to answer all queries, it must be possible to reconstruct the information needed to satisfy any query.

Analyses are repeatable. A classic technique for sequential program debugging is to re-execute the program with additional output statements, thereby providing more detail about the program execution. This technique depends on the fact that most sequential programs are deterministic, so that successive executions are essentially identical. Although parallel programs are not in general deterministic, any top-down methodology for parallel program analysis requires that information overlooked during analysis at an abstract level can be derived easily again.

Analysis is interactive. Although the collection and presentation of execution data can be automated, data interpretation must include feedback from the programmer. Only the information of current interest to the programmer should be provided. Also, the programmer should be able to shift the focus of interest at will.

The set of possible analyses is extensible. Any methodology limited to specific analyses will be unable to support the full development cycle for all parallel programs. The most common analyses, such as symbolic debugging, deadlock detection, contention analysis, and critical path analysis, obviously must be supported. However, since tool developers can't anticipate all the analyses that will prove useful, the methodology must also allow extensions that enable new, application-specific analyses.

This analysis methodology is rooted in basic principles that describe an effective approach to understanding complex systems. The effectiveness of cyclic debugging for sequential programs is no happenstance: it is effective because it is based on an analysis methodology that incorporates the criteria enumerated above.

The characteristics of the methodology described above motivate the approach to parallel program analysis taken in this dissertation. To support a top-down style of analysis, enough information must be recorded during a program execution to enable

[5] An example of an unreasonable query is "what statement was process *A* executing when process *B* executed statement 7?" Such queries require instantaneous snapshots of global state, which are impossible to capture in general, and are not particularly useful.

fine-grain detail to be recovered or recreated upon demand. This dissertation investigates techniques that support top-down analysis by recording execution traces based on partial orders of accesses to shared data. Each set of traces forms a compact characterization of a program execution and can be used to replay the execution upon demand.

These traces support top-down debugging based on repeated examination (*i.e.*, cyclic debugging); this approach enables reliable isolation of program faults. All reasonable questions about a program execution can be answered at any level of detail. In particular, fine-grain detail about a program execution can be recovered as needed during an execution replay without further distorting the execution under study. Analyses are repeatable, since the traces form a permanent record of the program execution. Analysis of traces, and execution replays based on these traces, is interactive, enabling programmers to focus the analysis. Finally, the program analysis tools developed to exploit these traces support an extensible set of analyses, providing an environment for development of special-purpose routines to analyze execution traces or control an execution replay.

Annotated with timing information, these traces also form an appropriate basis for top-down performance analysis of parallel programs. Using these traces, a programmer begins by analyzing global properties of a program execution such as communication cost, synchronization delay, processor utilization, and resource contention. As needed, the programmer can narrow the focus to equivalence classes of processes, or individual processes. The information contained in our execution traces enables analysis of the dynamic relationships between processes that are key to program performance. Fine-grain information detailing individual interactions between processes is available in the traces. Again, analyses are repeatable, since the traces form a permanent record of the program execution. Analysis of these traces is interactive. Initial analyses of a program execution at an abstract level are used to focus a programmer's attention on those aspects of the execution that merit more detailed examination. Finally, the tools developed for understanding program performance support an extensible set of analyses enabling the user to investigate application-specific issues. These tools can be extended to collect detailed information about the performance of code segments in individual processes during execution replay to facilitate tuning of the sequential code composing the processes.

1.4 Dissertation Organization

Chapter 2 reviews previous and ongoing research in debugging and analysis of parallel programs and their executions. Chapter 3 provides a formal foundation for execution replay techniques developed in later chapters. It presents a model of parallel program executions with shared-memory communication, defines precisely what criteria must be satisfied for two executions to be *indistinguishable*, and presents a theorem that establishes necessary and sufficient conditions for execution indistinguishability. Chapter 4 uses the results about execution indistinguishability in the development of techniques for minimal monitoring of program executions to provide execution replay. This chapter addresses practical issues in providing execution replay and presents performance implications of the resulting techniques. Chapter 5 describes an additional technique that

is needed to provide execution replay of programs that react to asynchronous events. Chapter 6 describes how synchronization traces can be used for debugging and performance analysis. It describes how to augment minimal synchronization traces to increase their utility, and describes the design and implementation of an integrated toolkit for debugging and analysis based on these traces. Chapter 7 presents some sample analyses constructed using a prototype of the integrated toolkit. Chapter 8 summarizes the contributions of this work, and provides some directions for future research.

2 Related Work

Techniques for debugging and analysis of parallel programs can be classified along two orthogonal dimensions: static techniques that focus on analysis of the source program itself, and dynamic techniques that focus on analysis of the execution behavior of programs. This chapter reviews a representative set of approaches for debugging and analysis of parallel programs along these dimensions.

2.1 Static Analysis

Static analysis techniques for parallel programs typically detect the existence of certain classes of software anomalies through analysis of synchronization primitives and variable usage information. Referencing an uninitialized variable and a dead definition of a variable are anomalies that are common to both sequential and parallel programs.[1] Such errors are readily detectable using data flow analysis. Possible concurrent access to a variable by conflicting operations (hereafter called a *parallel access anomaly*), and referencing a variable whose value is indeterminate[2] are anomalies that occur exclusively in parallel programs. For a restricted model of parallel programming that does not allow intertask synchronization (a task can only schedule execution of other tasks or wait for their completion), Taylor and Osterweil showed that each of these anomalies can be detected using data flow analysis based on process-augmented flowgraphs formed by connecting flowgraphs from each of the processes with synchronization edges that result from task scheduling operations [Taylor and Osterweil, 1980]. In addition, they demonstrated techniques to uncover anomalies associated with task scheduling using a restricted static tasking model.

Taylor found data flow techniques based on analysis of process-augmented flowgraphs to be inadequate for analyzing programs with intertask synchronization, such as the Ada rendezvous [Taylor, 1983b]. To cope with the difficulties introduced by intertask synchronization, Taylor developed analysis techniques based on the enumeration of all possible concurrency states of a parallel program, where each *concurrency state* is a tuple containing the *synchronization state* of each task in the system [Taylor, 1983b].

[1] A variable definition not subsequently referenced is called a *dead definition* [Osterweil, 1981, p. 245].

[2] The value of a variable is said to be indeterminate if there are two separate definitions for the value of the variable whose order of execution is indeterminate.

The synchronization state of a task corresponds to the last synchronization operation performed by that task. Taylor uses a concurrency history graph (CHG), composed of a node for each concurrency state and directed edges representing legal transitions between concurrency states, to represent all possible behaviors of a parallel program. A CHG for a program enumerates all rendezvous that can potentially occur during execution and all potential infinite waits (*e.g.*, deadlock). Furthermore, annotation of each edge in a CHG with the non-synchronization activities that occur between the two synchronization nodes that the edge connects enables potential parallel access anomalies to be detected by comparing actions on edges that are unrelated by sequential constraints.[3] Unfortunately, the complexity of performing these analyses on parallel programs with intertask synchronization is intractable in the general case [Taylor, 1983a]. Similarly, Callahan and Subhlok [Callahan and Subhlok, 1988] show that proving the absence of a parallel access anomalies in parallel programs without loops that use *post* and *wait* synchronization operations on events is Co-NP-hard.

Since static analysis of parallel programs is so costly in the general case, recent research has focused on techniques that reduce the cost for typical cases. Taylor [Taylor, 1983b] presents a technique for analyzing connected components of a concurrency history graph separately to reduce the analysis complexity, but he notes that the applicability of this technique is dependent on the style of concurrent programming (the programming style determines the size and number of connected components).

Most of the work in static analysis has focused on single-program-multiple-data (SPMD) programs in which a collection of identical tasks execute independently unless explicitly synchronized. Each task possesses a local data area, as well as access to a shared common data area. (For a more detailed description of the SPMD programming model, as well as other programming models for parallel processing, see [Karp, 1987].) The SPMD programming model includes programs written in the various dialects of parallel Fortran. Using Taylor's basic analysis framework, Appelbe and McDowell [Appelbe and McDowell, 1988a; Appelbe and McDowell, 1988b] describe a technique for reducing the size of CHGs for SPMD programs by collapsing sets of nodes in a CHG that represent a family of identical tasks into individual nodes. Although their technique results in smaller CHGs, they were not able to analyze real programs that used more than 6 tasks, since the CHGs still consisted of thousands of nodes. They conjecture, however, that for most SPMD programs, analysis of a small number of tasks should be sufficient to expose all anomalies detectable by this type of analysis [Appelbe and McDowell, 1988a].

Emrath and Padua [Emrath and Padua, 1988] present a novel technique for analyzing SPMD programs using *ordering graphs*. Each node in an ordering graph represents a program statement. There are two kinds of edges in an ordering graph: dependence edges, which indicate access conflicts between pairs of statements, and synchroniza-

[3]Taylor actually writes of augmenting each node n in the CHG with non-synchronization actions that occur along edges from n to each of n's successors [Taylor, 1983b, p. 374]. Actions along all edges originating from a common node can be grouped together after determining that none of the actions interfere or that the actions belong to edges that cannot occur in parallel (*e.g.*, the pair of edges represent different state transitions for the same process, as would occur with different choices of a *select* statement, or a different pairing of an *accept* statement with active entry calls).

tion edges, which indicate ordering relationships between statement instances that are guaranteed by use of semaphores [Dijkstra, 1968] and eventcounts [Reed and Kanodia, 1979]. Analysis of these ordering graphs determines if the dependences between statements are preserved by program synchronization. For typical SPMD programs in which statements using synchronization primitives are easily paired (*e.g.*, *await* and *advance* operations on the same eventcount), construction and analysis of ordering graphs seems to be an efficient technique for detecting (or showing the absence of) parallel access anomalies.

Static analysis techniques result in two types of error reports: *must* errors that occur for a program regardless of the execution path taken (*e.g.*, a variable whose value is always undefined before it is referenced), and *may* errors that occur along some execution paths, but not others (*e.g.*, a variable whose value is not defined along some, but not all, execution paths that lead to its use). A limitation of static analysis techniques is that they may report potential errors that can occur only along infeasible execution paths. This burdens the user with the responsibility of sifting out reports of real anomalies from those that could not possibly occur in any execution. Unfortunately, errors that occur only on infeasible paths cannot be distinguished from real errors without resorting to symbolic execution [Taylor, 1983b].

While static analysis techniques can be useful for locating stylized classes of errors in parallel programs (*i.e.*, parallel access anomalies, synchronization errors, and nondeterministic variable values), alone they are not a complete solution to the problem of developing correct parallel software.

2.1.1 Program Verification

Program verification is a more ambitious form of static analysis that attempts to identify semantic errors in a program. The intent of program verification is to prove that a program meets all of the criteria specified in some formal specification of its behavior. Many formal techniques for program verification have been studied (*e.g.*, [Babich, 1979; Lamport, 1977; Owicki and Gries, 1976; Rosen, 1976]); however, they are difficult to apply since they require the programmer to annotate the program with detailed assertions and invariants, which are often very subtle. There is considerable controversy about the role that program verification will play in computer science. De Millo, Lipton, and Perlis argue that "formal verification of programs ... will not play the same key role ... as proofs do in mathematics" [De Millo *et al.*, 1979, p. 271]. While most commentary following publication of this article [ACM, 1979] lauded the authors for their views, other commentary argued that verification is not a misguided endeavor as De Millo, Lipton, and Perlis claim. This controversy is by no means settled.

One of the fundamental limitations of program verification techniques is that even if they are successful, they can only determine if a program correctly implements an abstract specification; there remains the possibility that the specification does not adequately reflect the designer's intent. A recent article by Fetzer expresses a similar concern arguing that regardless of whether programs can be verified with respect to constraints impose by an abstract machine, verification is impossible for programs for

which there is an interpretation with respect to a physical system [Fetzer, 1988, p. 1059]. In order to discover that a specification and program is incomplete or otherwise incorrect, dynamic analysis of the program's behavior is necessary.

2.2 Dynamic Analysis

Dynamic analysis involves the collection of data about an executing program, interpretation of that data, and presentation of the digested information to the user. Both debugging and performance analysis are forms of dynamic analysis.

A central concern when developing tools and techniques for dynamic analysis is the effect that the data collection will have on the program (or system) under study. If the data collection is costly with respect to the granularity of the operations being monitored, the overhead of data collection will seriously distort the program execution. All software techniques for collecting dynamic information about a program execution affect the execution under study since the monitoring software competes with the executing program for hardware resources (the so called "probe effect" [Gait, 1985]).

Several researchers have proposed auxiliary monitoring hardware to enable non-intrusive monitoring (*e.g.*, [Brantley *et al.*; Rubin *et al.*, 1988]). Simple, special-purpose monitoring hardware can provide substantial benefit for collecting performance statistics at low cost (*e.g.*, [Brantley *et al.*]); however, hardware support for debugging involving any form of execution tracing is likely to be prohibitively expensive. Powell and Presotto's simulations of transparent message logging in a distributed system completely utilized one general-purpose processor to log interprocessor message-passing communication for five processors [Powell and Presotto, 1983]. Since tightly-coupled multiprocessor systems generally support faster communication than distributed systems, multiprocessors would require more monitoring hardware to cope with the greater data volume. For example, in the MAD system [Rubin *et al.*, 1988], Rubin, Rudolph, and Zernik expect to dedicate a special processor for monitoring for each processor in the machine.

Since comprehensive, non-intrusive monitoring requires so much additional hardware, it will be unavailable for most systems. Therefore, this dissertation focuses on software techniques for collecting execution traces of parallel program executions and accepts that these techniques will exert some effect on programs under study. An important design goal of software monitoring techniques is to minimize their effect on the program execution under study.

Techniques for dynamic analysis of parallel program executions can be classified into two broad categories: one-shot examination techniques and two-phase examination techniques. One-shot techniques are those that involve a single examination of an execution: all information about a state in the execution must be collected before leaving that state. Two-phase techniques involve collecting information that characterizes an execution during a monitoring phase, and using this information to repeatedly replay the execution to collect additional information as needed. Below, we provide an overview of these two approaches and explore their strengths and weaknesses.

2.2.1 One-Shot Examination

Since a parallel program may exhibit non-deterministic behavior, a particular execution path may not be readily repeatable. One-shot techniques do not address this issue directly; instead, they are used to gather any information desired about an execution in a single pass.

While one-shot techniques can be very useful for collecting performance statistics about an execution, they are not as useful for debugging parallel programs since they are incomplete for error diagnosis. Using a one-shot strategy to collect information for debugging forces a programmer to predict, *a priori*, the proper subset of information to monitor about a program execution to enable diagnosis of any error that might be observed. Unless the right information is collected during the execution, error diagnosis will be difficult, if not impossible. This is a troublesome restriction since the purpose of debugging is to diagnose unexpected flaws in a program; predicting what data will be relevant to diagnosing such flaws is very difficult.

One-shot monitoring techniques can be divided into two classes: passive and interactive. Passive monitoring involves specification of a finite set of tracepoints to be used for data collection before a program execution begins. Interactive monitoring subsumes the capabilities of passive monitoring, but also enables a program execution to be halted for interactive examination and modification of its current state, as well as insertion of additional tracepoints. Each class of techniques has its own advantages.

Passive monitoring strategies are important because they are generally less intrusive than their interactive counterparts and they are generally easier to implement than interactive strategies. One simple passive monitoring strategy is to insert print statements into key places in a program and examine their output; no additional tools are necessary.

McDaniel's Metric [McDaniel, 1977] is an early passive monitoring system for collecting information about distributed programs that communicate over a local area network. Metric has a tripartite structure: probes permanently inserted in the program under study that report information about program events, accountants that collect information from the probes, and analysts that interpret the information collected. Probes were designed for efficiency; they can remain in the system and not cause appreciable performance degradation or a substantial increase in network traffic. Debugging or performance analysis using this system uses post-mortem analysis of information collected from probes during a program execution. While ideally suited to the collection of performance statistics, debugging using data collected with Metric (or similar systems [Miller, 1985b; Garcia-Molina *et al.*, 1984]) is difficult, since only small amounts of information are collected; this limited data is likely to be insufficient for isolating the causes of a program failure.

Miller's DPM [Miller *et al.*, 1986; Miller, 1988] is similar in design to Metric. The primary difference between DPM and Metric is that Miller uses probes embedded in the kernel to monitor application programs, rather than inserting probes in the applications themselves. While DPM can be used on unmodified program binaries, Miller's system limits monitoring to events requiring kernel intervention. Such a technique could not

be used to monitor shared-memory communication. Miller's analysis techniques for the trace information include computing basic communication statistics at both process and system levels, measuring parallelism in the distributed computation under study (see [Miller, 1985a]) and computing a probabilistic causal relation between actions of processes. Also, Miller and Yang have explored techniques for hierarchical presentation of performance data for distributed programs [Miller and Yang, 1987], and techniques for presenting parallel program performance statistics based on critical path analysis [Yang and Miller, 1988]. As with Metric, the focus of DPM and IPS is on performance analysis; neither DPM nor IPS are designed for debugging.

Garcia-Molina, Germano and Kohler [Garcia-Molina *et al.*, 1984] propose a bottom-up method for debugging distributed systems.[4] Following preliminary module testing, in a distributed execution phase they record event traces for examination. This information is used to isolate an error within a single process. Since only incomplete information is recorded in the distributed execution phase, individual testing of the suspect process is necessary to pinpoint bugs. The primary advantage of separating debugging into two stages is that it reduces the volume of information that must be recorded during the distributed execution phase. The major disadvantage of this technique is the second stage requires construction of a detailed synthetic test environment that simulates all communication partners of the suspect processes.

To cope with the complexity of parallel program executions, Bates and Wileden [Bates and Wileden, 1983] propose a method of *Behavioral Abstraction* (BA). BA provides a mechanism for hierarchical description of events as sequences of *primitive events* that occur during program execution. These descriptions are used to present abstractions of program execution traces to a user. Hierarchical abstractions are useful, since they hide some of the underlying complexity of a parallel program; however, their fundamental disadvantage is that they require users to exhaustively describe interesting events in a program execution using a bottom-up specification. In creating such a specification, the user must anticipate all interesting events related to an error before execution; there is no mechanism for gathering additional information about an error after it is observed.

Using relational database techniques for storage and manipulation of program trace information has been studied as an alternative to the BA method for structuring execution trace information [Garcia-Molina *et al.*, 1984; Snodgrass, 1982; LeDoux and Parker, 1985]. The primary drawback of this class of methods is that relational processing of program events is slow. Two strategies have been examined for coping with this difficulty: limit the number of events traced during execution [Garcia-Molina *et al.*, 1984; Snodgrass, 1982], or slow the program execution to reduce the frequency of events [LeDoux and Parker, 1985; Linton, 1983], so that all events can be processed.

In general, interactive strategies are much easier to use than their passive monitoring counterparts. Passive strategies present the user with much unwanted and unnecessary information. Interactive strategies provide mechanisms for enabling and disabling

[4] Lauesen [Lauesen, 1979] defines bottom-up debugging as testing each individual module separately before putting them together and testing them as a system. In contrast, top-down debugging involves testing the whole system at once in nearly final form.

trace/breakpoints during the execution, avoiding collection of unnecessary information. Also, interactive strategies take advantage of context implicit in the user's focus of attention. Both of these capabilities provide abstraction of unwanted detail. Avoiding presentation of unnecessary information helps the user cope with the complexity of parallel program executions.

Typically, interactive multiprocess debuggers extend the facilities of single process debuggers to enable state-based examination of multiple processes (*e.g.*, [Redell, 1988; Weber, 1983]). While debuggers of this form can be very useful for debugging parallel programs with small-scale parallelism, the lack of mechanisms for directly monitoring process interactions makes debugging programs with large-scale parallelism difficult. At the other extreme, Helmbold and Luckham's debugger for monitoring Ada tasking [Helmbold and Luckham, 1984] neglects internal process state. Their implementation of a tasking monitor serializes task operations and thus is suitable for only single processor operation. Each of these systems provides access to only part of the information necessary to understand the behavior of parallel programs.

A significant disadvantage of interactive techniques is that they tend to lack the property of temporal transparency. Breakpoints used in interactive monitoring provide a mechanism for exerting unbounded intrusion into the delicate timing relationships existing between processes in a distributed computation. While halting a single process can be useful for detecting some sorts of errors, errors involving a relationship among multiple processes would likely require a more sophisticated investigation. In a distributed computation, it is not possible to instantaneously halt sets of processes that span multiple processors, since communication delays exist between processors. Halting distributed computations has been studied by several researchers [Cooper, 1987; Miller and Choi, 1986; Garcia-Molina *et al.*, 1984; Gait, 1985; Schiffenbauer, 1981], but none of the solutions adequately preserve temporal relationships in the general case.

Several interactive debuggers have been built that focus on the the problem of debugger intrusion into program executions. Schiffenbauer [Schiffenbauer, 1981] constructed a debugging system for distributed programs that communicate using a broadcast medium. Schiffenbauer's goal was to provide a temporally transparent interactive debugger that would support debugging of programs with real-time constraints. To provide transparency, Schiffenbauer's debugger requires that programs use a logical clock rather than a real-time clock. His debugger suspends processes whenever they send or receive messages. After completing whatever processing the communication operation required, the central communication manager awakens each blocked process after adjusting its logical clock. Although his system provides a measure of transparency, all communication must pass through a central node, which limits potential parallelism. Also, his solution for transparently maintaining logical time limits the distributed program to one process per node in the system. Cooper [Cooper, 1987] attempts to limit perceived debugger intrusiveness by halting all user processes on each node in the system along with each node's logical clock any time a process halts. Since all nodes cannot be notified and halted instantaneously, this too is only an approximate solution to the problem of debugger intrusiveness. To cope with the problem of debugger intrusiveness in Ada programs, DiMaio, Ceri, and Reghizzi [DiMaio *et al.*, 1985] constructed

an interpreter that simulates concurrent execution; in such an environment, debugger intervention is transparent to the executing Ada tasks. However, their technique is limited to multiprocess programs in a single processor environment and intimately tied to the semantics of the Ada language.

In contrast with most multiprocess debuggers, which support ad-hoc debugging methodologies, the MuTeam debugger for the ECSP language [Baiardi *et al.*, 1986] uses a formal specification of program behavior to monitor an execution for errors. Whenever a specification violation is detected, the debugger halts portions of the program and enables interactive examination of the executing program. The advantage of formal specification matching techniques is that they reduce the amount of information to be processed by a user. The primary problems with this approach are that debuggers using specification matching techniques exhibit unbounded intrusiveness into program executions while the specification matching occurs and that specification matching techniques require simultaneous debugging of the program and its specification. With the MuTeam debugger, an attempt is made to reduce the effect of debugger intrusiveness by providing a delay operator; however, this provides no real remedy. As with Behavioral Abstraction [Bates and Wileden, 1983], a drawback of specification matching approaches is that specifications must exhaustively describe permissible behaviors for it to be effective in detecting program errors. Other specification matching techniques for monitoring program behavior including path expressions [Bruegge and Hibbard, 1983], and temporal logic [Harter *et al.*, 1985] suffer from the same drawbacks.

Linton [Linton, 1983] and Snodgrass [Snodgrass, 1988] propose using relational queries to control execution of a program under study. Users pose relational queries about a program execution and these queries enable trace/breakpoints during a subsequent execution of the program to collect information to satisfy the query. This model provides a view of a program execution as a historical database without actually constructing the database. While this approach seems promising, little practical experience has been gained using such a relational model.

2.2.2 Two-Phase Examination

Two-phase strategies for parallel program debugging have evolved as a combination of passive and interactive methods. During the first phase, passive monitoring techniques are used to record a characterization of a parallel program execution. It is important to note that these execution characterizations do not contain complete information about program executions, as do those in Balzer's EXDAMS system [Balzer, 1969] for debugging sequential programs. Recording a complete history of a parallel program execution would be extremely costly in both time and space. Instead, only partial information is collected; internal changes to the state of a process are not typically recorded. This partial information is used during a second phase to reproduce parts of the execution for further study, creating additional detail on demand. The replay phase generally has an interactive flavor that enables a programmer to use analysis tools to interrogate an execution for further information. The principal advantage of two-phase techniques is that they readily support the familiar cyclic debugging strategy used for sequential programs. Two-phase techniques have been studied in several domains, including

loosely-coupled systems in which processes communicate via messages, systems based on nested atomic transactions, and concurrent programs that use semaphores and monitors for synchronization and communication.

Methods to reproduce the execution behavior of programs comprised of loosely-coupled processes that communicate using messages typically require that the contents of each message be recorded in an event log as it is received [Curtis and Wittie, 1982; LeBlanc and Robbins, 1985; Smith, 1984]. The programmer can either review the events (messages) in the log, in an attempt to isolate errors, or the events can be used as input to replay the execution of a process in isolation. There are several disadvantages to this approach. First, it has only been used in loosely-coupled systems and it would not be well-suited to tightly-coupled systems. Although the amount of data exchanged in messages could be very large, this technique exploits the fact that communication in loosely-coupled systems takes place infrequently, primarily because of the high cost of communication. The additional time necessary to copy a message into an event log in local memory does not seriously affect performance when compared with the time required to send a message. This assumption does not apply to tightly-coupled systems, where the cost of communication is lower, allowing more frequent communication. Another disadvantage is that the space requirements for the event log tend to be very large. Again, within the domain of loosely-coupled processes, it is reasonable to assume the logs will grow slowly enough that they can be buffered in memory and then stored on external devices without seriously affecting the performance of the program. In spite of these drawbacks for tightly-coupled systems, Pan and Linton [Pan and Linton, 1988] recently proposed a system that logs shared-memory communication, as well as messages. They expect that their system will need to cope with data volumes of at least 1Mb per process per second [Pan and Linton, 1988, p. 127]. Such an approach will likely be impractical for large-scale systems. The third, and most important drawback of these data logging approaches, is that it is difficult to examine the global effects of process interactions using this technique, since the replay mechanism only operates on a single process in isolation. Previous attempts to replay groups of processes using this scheme require that a network-wide consistent time be maintained [Curtis and Wittie, 1982].

Powell and Presotto used a similar message-logging approach to provide fault tolerance [Powell and Presotto, 1983]. Their system collects message histories from a broadcast network to enable computations to be restarted in the case of failure; however, they envisioned using these logs for replaying processes for debugging as well. A limitation preventing extension of their method for general purpose multiprocess debugging is that it relies on the use of a broadcast communication medium for all communication. In addition, their technique apparently does not scale: simulations predict that their system cannot cope with message traffic from more than five communicating processors.

Chiu's technique for replaying a program's execution in an atomic transaction system involves checkpointing each version of all atomic objects and recording a timestamp for each atomic action during program execution [Chiu, 1984]. A debugger uses this information to traverse action trees (corresponding to the nested atomic actions of a program execution) according to a serialization of their constituent atomic actions. Traversing

an action tree permits viewing the state of atomic objects before and after each atomic update, as well as replaying execution through action sequences to isolate program flaws. A similar technique has been described by Lin and LeBlanc [Lin and LeBlanc, 1988] for debugging object/action programs in the Clouds system. Although these techniques require significant storage overhead to maintain the necessary checkpoints of atomic objects, the checkpoints may be required for recovery actions anyway.

In contrast to the event logging and checkpointing approaches that record all data shared between processes, sequencing methods record small amounts of ordering information that relate program events. This information provides enough detail to enable replay of parallel computations for further study. During this replay, values of data shared between processes are recreated as needed. Sequencing methods offer the most promise for monitoring parallel computations since, in general, they record significantly less information about program executions than data-logging strategies. This reduction of information recorded has two effects: (1) monitoring tools have reduced needs for storage and (2) monitoring overhead is significantly lower than for data-logging strategies and therefore, sequencing techniques are more temporally transparent.

Carver and Tai have considered repeatable execution for programs consisting of concurrent processes that interact through semaphores and monitors [Carver and Tai, 1986]. In their approach, an execution of a concurrent program is characterized by a sequence of P operations (termed a *P-sequence*) on shared semaphores. A P-sequence is a sequence of ordered pairs; each pair corresponds to a P operation on a specific semaphore by a specific process. Thus, a P-sequence is a total order of *all* synchronization operations that occur in a program. P-sequences can be created by the programmer to test specific synchronization sequences of a concurrent program or can be recorded during execution to provide repeatable execution. The same idea can be used to produce an *M-sequence* for monitors, which records a sequence of calls to all monitor entry procedures. The disadvantage of this approach is that it requires that all P operations be serialized, thereby losing much of the potential for parallelism that exists in a program. While adequate for single processor systems that simulate concurrency, these techniques would not be useful for testing programs that use multiple critical sections in a parallel environment. There, the serialization constraint would have such an impact on program performance that it would be impractical to monitor programs during normal execution.

2.3 Combined Techniques

Recently there has been considerable interest in combining static and dynamic analysis techniques.

Miller and Choi [Miller and Choi, 1988] present a technique for using flowback analysis for parallel program debugging. Their technique is similar to Balzer's [Balzer, 1969], which enables a programmer to browse forward and backward in an execution to determine when particular changes to the program state occurred. Although this is a powerful model for debugging, the primary drawback of flowback analysis is the huge volume of data that has to be recorded during execution. Instead of recording all changes to a program's state during execution, Miller and Choi use interprocedural and data flow

analysis to determine a subset of the data and control flow information that must be recorded during an execution monitoring phase. In a debugging phase, their technique for flowback analysis uses the recorded information to recreate data values as needed to respond to user queries. Miller and Choi claim that this technique will reduce the volume of information recorded for flowback analysis to a manageable amount; however, it appears that the information they record may still be voluminous. In a prologue for each block of code in a program, they record the values of all shared variables that the block references. Similarly, in an epilogue for each block of code, they record the values of all variables that are modified by the block. Since all communication in shared-memory parallel programs arises through access to shared variables, Miller and Choi's technique involves recording all values communicated between processes. Therefore, the amount of data recorded using their techniques should comparable to that recorded using message-logging techniques.

Emrath, Allen and Padua [Allen and Padua, 1987; Emrath and Padua, 1988] propose a three phase approach to debugging parallel Fortran programs. The three phases of their approach are static analysis, run-time tracing, and trace analysis. When their techniques for static analysis fail to produce conclusive results regarding the presence or absence of parallel access anomalies, they fall back on analysis of dynamic execution traces. They propose instrumenting programs to record information about memory references and synchronization operations. Parallel access anomalies (conflicting accesses to the same memory location that are unordered by synchronization) are detected by using the dynamic trace information that they collect to build synchronization graphs annotated with memory reference information.

Young and Taylor [Young and Taylor, 1988] propose integrating static analysis and symbolic execution. Their approach uses static concurrency analysis as a path selection mechanism for symbolic execution, and uses symbolic execution to prune infeasible concurrency states for static analysis. By combining these two techniques they aim to suppress reports of errors that occur along infeasible execution paths, as well as reduce the number unreachable concurrency states that are explored.

These last two approaches both inherit the limitations of static analysis techniques; they are capable of detecting only a stylized class of errors related to variable usage. Dynamic analysis techniques involving programmer interaction are necessary to uncover other types of errors.

2.4 Relationship to Other Work

The work described in this dissertation differs from other research in parallel program analysis in two key respects. First, it focuses on debugging and analysis of programs executing on large-scale, shared-memory multiprocessors. Shared-memory multiprocessors present some unique challenges for execution monitoring and debugging. Second, our approach possesses a degree of integration between parallel program debugging and performance analysis not found in existing systems.

Unlike the systems of Schiffenbauer [Schiffenbauer, 1981] and Cooper [Cooper, 1987], our approach to approximating temporal transparency does not attempt to mask the

intrusion of interactive debugging into program executions. Instead, we use a two-phase strategy for execution analysis. During an initial monitoring phase, minimal information that characterizes a program execution is recorded; interactive examination of the program execution is supported in a subsequent analysis phase. By separating interactive analysis from the execution under study and recording only minimal information, we disturb the execution as little as possible and minimize the probe effect.

Our approach to two-phase analysis is most similar to that of Carver and Tai [Carver and Tai, 1986] in that we record only information about synchronization events between processes rather than recording all data communicated between processes [Chiu, 1984; Curtis and Wittie, 1982; LeBlanc and Robbins, 1985; Pan and Linton, 1988; Smith, 1984]. However, unlike the monitoring technique used by Carver and Tai, our synchronization tracing technique is fully distributed, making it more appropriate for monitoring programs executing on parallel hardware.

Our work is concerned with both debugging and performance analysis. In the sequential programming world, the profiler *gprof* [Graham *et al.*, 1982], and the symbolic debugger *dbx* both use the source code as a common representation, but lack integration through a common user interface. Both the PIE parallel programming environment [Segall and Rudolph, 1985], which uses an integrated relational database view, and IPS [Miller and Yang, 1987], which uses an integrated computation hierarchy, possess a desirable degree of integration. However, both systems emphasize performance analysis; debugging is a secondary consideration. The Jade system [Joyce *et al.*, 1987] uses a common representation, called an event trace, as the basis for a toolkit, but the tools are not integrated. For example, a textual console is used in Jade to describe a sequence of events, but a graphical display is used to describe the current state of the execution. Our approach to debugging and performance analysis is based on a common representation of program executions, a methodology for analyzing executions, and an integrated toolkit that supports the methodology.

3 Modelling Parallel Program Executions

This chapter presents a formal model of parallel program executions in which processes communicate using shared memory. Results proven using this model serve as a basis for designing execution monitoring techniques that support a top-down approach to debugging parallel programs executing on shared-memory multiprocessors.

Isolating the cause of an error in a parallel program execution involves examining the sequence of state transitions for each process. To do this in a top-down manner, a programmer must be able to examine the process states in a program execution repeatedly, recovering greater detail as needed in successive examinations. Two-phase techniques that provide execution replay support this top-down style of analysis. To provide replay of parallel program executions in loosely-coupled systems, message-logging techniques have been used to record all inputs to each process. However, on shared-memory multiprocessors, the volume of data that crosses process boundaries during executions of large-scale parallel programs can be overwhelming, making such data-logging techniques impractical [LeBlanc and Mellor-Crummey, 1987].

This chapter explores an alternative to using data-logging techniques to provide replay of parallel program executions. In the context of an abstract model, the proofs in this chapter demonstrate that if certain synchronization relationships are maintained between processes in each of a pair of executions of a parallel program, the resulting executions will be *indistinguishable*. Indistinguishable executions, from the standpoint of program debugging, are those in which corresponding processes go through the same sequence of states. The ability to create indistinguishable executions supports the use of a top-down approach for debugging based on repeated examination (*i.e.*, cyclic debugging). To isolate an error in a program execution using a cyclic debugging strategy, it is not necessary that the execution examined in each iteration of the technique be identical to the original, rather it suffices if the execution is merely indistinguishable from the original.

The proofs in this chapter provide necessary and sufficient conditions for executions of parallel programs programs to be indistinguishable under our model. The final section of this chapter briefly describes how these results form the basis for a practical execution monitoring technique that collects sufficient information to enable indistinguishable executions of parallel programs to be created.

3.1 A Formal Model for Shared-Memory Programs

This section presents a model for parallel program executions that use shared-memory communication. A parallel program execution is composed of a set of processes, each of which executes its own sequence of instructions asynchronously with respect to other processes (*i.e.*, there is no global clock in the system and no assumptions may be made about the relative speed of processes). In our model, all interprocess communication is accomplished by reading and updating values in a common global memory. The model we present here draws elements from a model of asynchronous distributed systems by Panangaden and Taylor [Panangaden and Taylor, 1988], which avoids using an explicit notion of time, and a global state model of distributed systems by Chandy and Lamport [Chandy and Lamport, 1985]. In both of these models, however, processes communicate using message passing.

To keep the shared-memory model of communication simple, all memory elements are assumed to be accessible to all processes in an execution. To ensure that the communication model is realistic (*i.e.*, memory elements are single-valued entities with persistent state), we define memory elements in terms of a temporal *value trace.*

Definition 1 *A* value trace *of a memory element* m *is a (possibly infinite) sequence of values beginning with the initial value 0. Such a sequence is written as follows*

$$t_m = 0, v_1^m, v_2^m, v_3^m, \ldots$$

Each value v_j^m *in* t_m *is drawn from a finite set of values* $MemVal = \{0, \ldots, N\}$ *that any memory element may assume.*

Before we can explain how transitions in a value trace occur, we need to describe *processes*, the active agents that operate on memory elements.

Definition 2 *A* process *is defined by*

1. *A finite set* S *of states. Each state* $a \in S$ *specifies a reference to at most one memory element* m_a *and a fetch-and-*Φ *[Gottlieb and Kruskal, 1981] read-modify-write operation* Φ_a *to apply to the contents of* m_a.

2. *An initial state* $s_0 \in S$.

3. *A transition function* δ *that maps each element of* $S \times \{MemVal \cup null\}$ *to at most one successor. More precisely,* $\delta(s, v(m_s)) = (s', \Phi_s(v(m_s)))$ *maps a state* $s \in S$ *and* $v(m_s) \in MemVal$*, the current value of memory element* m_s *whose reference is specified by* s*, to a successor state* $s' \in S$ *and* $\Phi_s(v(m_s)) \in MemVal$*, a new value for* m_s*. If the state* s *does not specify a reference to any memory element, the distinguished value* null *is used in place of* $v(m_s)$ *and* $\Phi_s(v(m_s))$.

Since δ maps each state and memory value pair into at most one new state and memory value pair, processes are deterministic with respect to their inputs and current state.

A *computation step* of a process can be described as a quadruple $\langle s, s', v, v' \rangle$. The elements of the quadruple are s, the state of the process before the step, s', the state of the process after the step, v, the value of m_s before the step and v', the value of m_s after the step. If no memory location is referenced in state s, v and v' are the distinguished value *null*. For each process, the transition function defines the legal computation steps. There are two types of computation steps: self-contained steps, which do not reference any memory element, and steps that indivisibly read and update the value of a memory element using a read-modify-write operation. A step that accesses a memory element m_s is either a *read* operation, which leaves the value of m_s unchanged (Φ_s is the identity function), or an *update* operation, which may modify the value of m_s (Φ_s is not the identity function).

Armed with a definition of processes, we can describe how they manipulate memory elements, resulting in value traces. Each transition in the value trace of a memory element is caused by an *update* operation on the memory element. A value trace defines a total order of *update* operations on a memory element. Although *read* operations on a value are implicitly ordered with respect to the *update* operations that affect that value, *reads* of the same value are unordered; thus, our model of memory elements explicitly represents concurrent-read-exclusive-write semantics. Our model requires explicit specification of a legal sequence of *updates* for each memory element in the form of a value trace to avoid the need for complex axioms to describe the legal behavior of memory elements (in particular, to ensure that memory elements don't spontaneously change value).

The potential executions of a process can be described as a finite set of *local histories.*

Definition 3 *A* local history *of process i is a (possibly infinite) sequence of computation steps. Such a sequence is written as*

$$h_i = \alpha_0^i, \alpha_1^i, \alpha_2^i, \alpha_3^i, \ldots$$

where α_j^i is computation step j of process i. The first component of α_0^i must be s_0^i, the distinguished initial state for process i.

A *program* is a finite, ordered set of processes. A process in a program is referred to by its index in the set. Defining a program as an ordered set simplifies the notation for comparing multiple executions of the same program.

A *program execution* of a program P is specified by a triple $\langle H, V, M \rangle$ where H is a set of local process histories (one for each process in P), V is a set of value traces (one for each memory element), and M is an *access mapping* that relates the computation steps in H to values in V, the memory element value traces. An access mapping is a set of quadruples of the form $\langle p, i, m, j \rangle$, where p designates a process, i is the index of the computation step α_i^p in h_p, m is the memory element referenced during α_i^p, and j is the index of the value v_j^m that is referenced in the value trace t_m. Each computation step in H that references a memory element must have a single corresponding quadruple in M. There is a bijective mapping between computation steps in H that update a memory element and values of index > 0 in the value traces V. A bijection is necessary

so memory element values do not spontaneously change, and the effect of every *update* operation is noticed. Below we describe an additional constraint that must be met for a triple to specify a feasible program execution.

Although our execution model contains no explicit notion of time, there is a loose ordering of computation steps in the system that can be expressed using a variant of Lamport's *happened-before* relation [Lamport, 1978]. The happened-before relation expresses potential causality, which links two computation steps if it is possible for one to have an effect on the other. For a parallel program using shared-memory communication, potential causality results from sequential execution of single processes and process interactions through shared memory.

Definition 4 *A computation step α_1 happens before α_2, denoted by $\alpha_1 \rightarrow \alpha_2$, iff one of the following conditions is true:*

1. *α_1 and α_2 are both in the local history of the same process and α_1 occurs first in the sequence.*
2. *α_1 and α_2 both access the same memory location m, M maps α_1 to value v_j^m in t_m, M maps α_2 to v_k^m in t_m, and $j < k$.*
3. *α_1 and α_2 both access the same memory location m, M maps both α_1 and α_2 to v_j^m in t_m, α_1 is an* update, *and α_2 is a* read.
4. *there exists a computation step α_3, such that $\alpha_1 \rightarrow \alpha_3$ and $\alpha_3 \rightarrow \alpha_2$.*

For $E = \langle H, V, M \rangle$ to specify a feasible program execution, $\rightarrow_E$, the happened-before relation for execution E, must be a partial order. The precedence constraints on computation steps (as defined by the happened-before relation) result from the total order of each local process history and the partial order induced by mapping computation steps to values in the ordered value traces. These precedence constraints must be acyclic; a cycle in these constraints would mean that a value of a memory location is read before it is written, or that an old value of a memory location is read after it is overwritten.

3.2 Conditions for Execution Equivalence

Definition 5 *Two program executions, E and E', are said to be* indistinguishable *to a process i if both executions assign the same sequence of computation steps to process i. The indistinguishability of two program executions E and E' with respect to process i is denoted as $E \sim_i E'$.*

Intuitively, if two executions of a program are indistinguishable to a process i, then process i goes through the same sequence of state transitions and sees the same sequence of values in memory in both executions. This definition of indistinguishable behavior of an individual process forms the basis for a definition of indistinguishability for program executions:

Definition 6 *Two program executions E and E' are indistinguishable iff $\forall_i\ E \sim_i E'$. We denote that two program executions are indistinguishable as $E \approx E'$.*

Indistinguishability provides a sensible notion of program execution "equivalence" for state examination based debugging techniques, since each process passes through the same sequence of states in a pair of indistinguishable executions.

Below, we prove necessary and sufficient conditions for two executions of a program to be indistinguishable.

Lemma 1 *Two executions $E = \langle H, V, M \rangle$ and $E' = \langle H', V', M' \rangle$ of a program P are indistinguishable if their access mappings M and M' are identical.*

Proof: Suppose two M and M' are identical, but E and E' are distinguishable. There must be at least one pair of corresponding local histories in E and E' that are distinguishable. Let T be any topological sort of the computation steps in E' consistent with the happened-before relation $\rightarrow_{E'}$. Let α' be the first computation step in the sorted order T that differs from its corresponding computation step α in E.[1] Let h'_i be the local history of the process to which α' belongs and j be the index of α' in h'_i (the corresponding step α must be step j of h_i).

By the definition of a process transition function, for α and α' to differ, α' must either start in a different state from α, or the value of the memory element seen in step α must differ from the value of the memory element seen in step α'.

However, α and α' start in the same state as shown by the following two cases:

1. $j = 0$: Since E and E' are both executions of the program P, process i has the same definition for both E and E'. By definition, a process has a single initial state, thus α' and α start in the same state.

2. $j > 0$: Step $j-1$ in h'_i must be identical with step $j-1$ in h_i since

   - step $j-1$ in h'_i precedes step j (*i.e.*, α') in $\rightarrow_{E'}$ (case 1 of definition 4),
   - T preserves $\rightarrow_{E'}$,
   - and by the hypothesis, α' is the first computation step in T to differ with its corresponding step in E.

   Since step $j-1$ in h'_i and h_i are identical, they must end in the same state. Therefore, since each computation step with index $j > 0$ in a local history starts in the state in which the previous step ended, α' and α start in the same state.

Since α' and α start in the same state, call it s, then the only way α' and α can be different is if they see different values when referencing a memory element. We must consider two cases:

[1] Corresponding computation steps are trivial to identify; they are steps at the same index in the local histories of corresponding processes. By definition, corresponding processes have a common index in the program P, an ordered set of processes.

1. State s does not specify a reference to any memory element. Since α' and α belong to the same process, by the definition of a process transition function, α' and α must be identical, which contradicts the original supposition.

2. State s specifies a reference to memory element m_s and an operation Φ_s.

   Any *updates* to m_s will be related to α' (respectively, α) by M' (respectively, M) since M' (M) defines a total order for the sequence of *update* operations on each memory element (cases 2 and 4 in definition 4). The last *update* to memory element m_s that precedes α' in M' (if any such *update* exists), must be the last *update* to precede it in T. This follows since T preserves $\rightarrow_{E'}$, which defines a superset of the ordering constraints defined by M'. Since M is identical to M', the indices of the computation steps that precede α' in M' (and therefore in T) are identical to the indices of the computation steps that precede α in M (likewise, in E). By hypothesis, α' is the first computation step (of execution E') in T to differ with its corresponding computation step α in E; therefore, all of the computation steps that precede α' in T (including all those that precede α' in M') must be identical to those that precede α in E (including all those that precede α in M). On this basis, we may conclude that the last *update* to memory element m_s that precedes α' (if any such *update* any exists) in E' (and therefore in M') is identical to the last *update* to memory element m_s that precedes α (if any such *update* exists) in E (and therefore in M). Since the last *update* that precedes α' must be identical to the last *update* that precedes α, α' and α will see the same value of m_s and thus be identical. This contradicts the supposition that α' and α are different.

Since α and α' were shown to be identical in all cases under the condition in the Lemma, the supposition that there exists some α' in E' that differs from the corresponding step α in E is proven false. Therefore, $E \approx E'$. □

Lemma 1 shows that for a given program, if the access mappings in a pair of executions are identical, the executions are indistinguishable. Coupled with the happened-before relation, an access mapping defines a total order of *update* operations for each shared-memory element, and a total order of *reads* with respect to updates. In the terminology used for describing data dependences in optimizing compilers, the condition in Lemma 1 amounts to requiring that the *flow dependences*, *antidependences*, and *output dependences* that exist between computation steps in the execution E are preserved in E'. (For a description of flow dependences, antidependences, and output dependences, see [Padua and Wolfe, 1986, p. 1185].)

The intuition behind Lemma 1 is that if a deterministic process is supplied *the same input values* (corresponding to the values of shared memory elements referenced) *in the same order* during successive executions, it will produce the same behavior each time. In particular, the process will produce the same output values in the same order. Each of those output values may then serve as an input value for some other process. Therefore, if the interleavings of processes in two executions of the same program are such that the processes operate on shared memory elements in the same order (with the exception that the order of a set of *reads* of the same value of a memory element

by different processes is unimportant) with respect to each memory element, all of the processes will see the same input values and produce the same output values, resulting in indistinguishable executions.

Lemma 2 *There exists a program P such that two executions $E = \langle H, V, M \rangle$ and $E' = \langle H', V', M' \rangle$ of P are indistinguishable only if M and M' are identical.*

Proof: Consider a program P with $2k$ processes. For each process $1 \leq j \leq 2k$, let process j consist of $k+2$ states, an initial state s^j_{init}, and $k+1$ final states $s^j_{final_i}$ for $0 \leq i \leq k$. Each process references the memory element m_0 in its initial state s^j_{init}; processes $1 \leq j < k$ replace $v(m_0)$, the value of m_0, with $\Phi(v(m_0)) = j$ and processes $k < j \leq 2k$ read $v(m_0)$, leaving it unchanged. For processes $1 \leq j \leq k$, $\delta(s^j_{init}, i) = (s^j_{final_i}, j)$, for $0 \leq i \leq k$. For processes $k < j \leq 2k$, $\delta(s^j_{init}, i) = (s^j_{final_i}, i)$.

Suppose there exist two executions E and E' of the program P such that $E \approx E'$, but M and M' are not identical. Two cases must be considered:

1. M' does not preserve the total order of updates specified by M. If M' does not preserve the total order of updates specified by M, there must exist some pair of updates α and β in E such that β immediately follows α and corresponding updates α' and β' in E such that β' occurs before α'. Recall that each update computation step specifies both an input value from m_0 and a subsequent output value for m_0. The input value from m_0 for β must be the final value of m_0 from α since the value of memory elements only changes in response to update operations. By definition of the program P, each update operation leaves a unique value in m_0; therefore, since β' precedes α', the preceding value of m_0 cannot be the final value of α', thus β and β' cannot be identical. Since β and β' differ, the pair of process histories that contain them are distinguishable. Therefore, E and E' are distinguishable contradicting the supposition.

2. M' does not preserve the total order of reads with respect to updates specified by M. Consider the case in which a *read* α precedes an *update* β in E and the corresponding *read* α' immediately follows β' in E'. α' will see the value that β' wrote in m_0 in E', since the value of memory elements only changes in response to update operations. However, since each pair of corresponding *updates* in E and E' writes a different value in m_0, α could not see the value written by β. Therefore, α and α' cannot be identical. As above, since α and α' differ, the pair of process histories that contain them are distinguishable. Therefore, E and E' are distinguishable contradicting the supposition. The symmetric case in which *read* α immediately follows *update* β in E and the corresponding *read* α' precedes *update* β' in E' follows similarly.

Therefore, no two indistinguishable executions of E and E' of the program P exist for which their access mappings M and M' are not identical. □

Theorem 1 *Given a program P and an execution $E = \langle H, V, M \rangle$, to produce an execution $E' = \langle H', V', M' \rangle$ that is indistinguishable from E, it is necessary and sufficient to ensure that the access mappings M and M' are identical.*

Proof: Lemma 1 shows that if the access mappings M and M' are identical, then $E \approx E'$. Lemma 2 demonstrates the existence of a program for which any two executions E and E' are distinguishable unless M and M' are identical. □

3.3 Practical Applications

Theorem 1 states that for an abstract model of shared-memory parallel program executions, an indistinguishable execution can be constructed by ensuring that certain ordering properties of the computation steps in the original execution are preserved. Thus, if information about the order in which processes access each memory element can be recorded during an execution and an equivalent order can be enforced during subsequent executions, then indistinguishable executions can be reproduced on demand. As stated earlier, the ability to create indistinguishable executions on demand is useful since it enables use of a top-down debugging strategy based on repeated examination.

Clearly, if one applied the result of Theorem 1 directly and recorded ordering information at the level of individual memory accesses for a real program executing on a shared memory parallel processor, the volume of data recorded for the execution would be overwhelming and the action of recording the information would greatly distort the execution. However, for parallel programs that share data at a coarser grain, the result of Theorem 1 forms the basis for a practical monitoring technique, as described in the next chapter.

4 Deterministic Replay of Parallel Program Executions

The previous chapter presented an abstract model of parallel program executions and proved necessary and sufficient conditions for executions to be indistinguishable. Here, we use those results as the basis for a minimal monitoring technique for shared-memory parallel programs. We present a general solution for reproducing the execution behavior of parallel programs, focusing on providing repeatable execution of large-scale parallel programs in tightly-coupled systems.

During a program execution, we save the relative order of accesses to shared data structures, not the data associated with each access. Since the data values communicated between processes (communication is mediated by the use of shared data structures) are not saved, as they are by data-logging approaches [Chiu, 1984; Curtis and Wittie, 1982; LeBlanc and Robbins, 1985; Pan and Linton, 1988; Powell and Presotto, 1983; Smith, 1984], this approach requires less time and space to save an execution trace supporting program replay than other methods, provided that the interactions with shared data structures are not too fine-grain. This property makes our technique especially useful for monitoring parallel programs on tightly-coupled multiprocessors, where interprocess communication is cheaper, and therefore used more frequently than in loosely-coupled systems.

The information recorded in these traces enables program behavior to be reproduced upon demand during the debugging cycle. Indistinguishable executions are created by providing the same input from the external environment and imposing the same relative order on events during replay that occurred during the original execution.

Unlike previous techniques, our execution tracing technique is independent of the particular form of interprocess communication used. This is important since shared-memory multiprocessors can support a wide variety of communication and synchronization abstractions, including direct use of shared memory, message passing, and remote procedure call. Also, during monitoring and replay, we avoid introducing any global synchronization of events through the use of a fully distributed protocol; there is no centralized bottleneck and no need for synchronized clocks or a globally-consistent logical time.

In the following sections we explore practical issues in reproducing the execution behavior of parallel programs. Section 4.1 examines the need to simulate the external environment to accurately replay a program execution. Section 4.2 describes in detail

a technique for collecting minimal synchronization traces and using them to provide repeatable execution of parallel programs. Finally, section 4.3 describes a prototype implementation on our BBN Butterfly, a tightly-coupled multiprocessor comprised of 128 MC68000 processors.

4.1 Simulating the External Environment

As with any cyclic debugging technique, we assume that the original execution of a program and subsequent replays occur in equivalent virtual machine environments. Two virtual machines A and B are said to be equivalent with respect to program P if program P can exhibit the same behavior whether executed on virtual machine A or B. For practical reasons, we do not require equivalent physical machine states, since that would include the contents of all external devices, the exact value of the clock, and the internal states of all components. In particular, A and B need not have identical real-time clock values if P's execution does not depend on the real-time clock. Similarly, the contents of file F on machine A and B can differ if P does not attempt to reference F. If program P depends on physical details of its virtual machine during execution, it becomes difficult, if not impossible, to simulate the virtual machine during replay.

Real-time programs, in particular, cause difficulties for simulating equivalent virtual machines. We require that programs receive identical input from the environment during both execution and replay. However, it is not sufficient simply to supply the same input to the process, we must also supply it at the same points during program execution. This can be difficult for real-time programs since the arrival of input is often signalled by asynchronous interrupts. Without making special provisions to record when interrupts occur during program execution, we cannot accurately simulate the original virtual machine environment. We defer this issue until the next chapter, where we present a technique for pinpointing the occurrence of asynchronous events during a program execution. For the remainder of this chapter, we assume that the program executions under study do not rely on the use of asynchronous primitives.

It is important to note that the problem of finding equivalent virtual machines also arises when debugging sequential programs; it is orthogonal to the specific problem of debugging parallel programs. We do not depend on a particular simulation of virtual machines, so any techniques developed for sequential program debugging can probably be used. Specifically, we assume that programs do not exploit the physical characteristics of any resources allocated by the system, therefore, we need only ensure that the amount of resources available during replay is at least the amount used by the program during the original execution. Any unsuccessful attempt to allocate resources during execution can be recorded, so that the same behavior can be re-created during replay.

4.2 Synchronization Traces for Program Replay

Theorem 1 in chapter 3 shows that if in two executions of the same program, the processes interleave in such a way that the order of *updates* to each shared memory

element is identical, and the order of *reads* with respect to *updates* is identical (both of these conditions are implied by having identical access mappings), then the program executions will be indistinguishable. Here, we apply this result to develop a practical execution tracing technique that records sufficient information to enable creation of indistinguishable executions to simulate execution replay.

4.2.1 Communication Through Shared Objects

Although it is impractical to trace the order in which processes access individual shared memory elements during a program execution (due to the volume of trace information that would be generated), the result of Theorem 1 still applies if we consider data sharing at a coarser granularity. In our approach, all interactions between processes are modeled as operations on shared objects. This characterization of process interactions is not restrictive since all communication and synchronization primitives can be reduced to operations on shared data. Typically, each shared object corresponds to an instance of an abstract data type such as a mailbox, a message buffer, a row in a matrix, or a monitor. Most sharing relationships that occur in MIMD parallel programs occur at this granularity.

Our approach exploits the fact that values exchanged between processes via shared data depend only on the initial values in shared objects, the order in which processes are granted access to the shared objects, and the deterministic nature of processes. As in our model of shared memory elements in chapter 3, operations on shared data objects can be separated into two classes: *read* operations, which do not change the state of an object, and *update* operations, which may. For an execution interleaving to be readily repeatable, we require that the set of operations on each shared object be linearizable. A set of operations is linearizable if the result of each individual operation is the same as it would be if the operations had all been executed in some legal sequential order that is consistent with the real-time relationship between the operations. If a set of overlapping operations is not linearizable, a record of the interleaving of processes at the level of individual memory accesses would be necessary to enable creation of an indistinguishable execution. A protocol that ensures a valid linearization, such as a concurrent-read-exclusive-write (CREW) protocol [Courtois *et al.*, 1971], must be used for access to each shared object. In choosing a protocol, we look for one that guarantees linearizability, while exerting minimal impact on shared object access and allowing maximal parallelism. If an access protocol that guarantees linearizability for operations on shared objects is already present in the application or the system, it is not necessary to superimpose another. Therefore, our techniques are applicable to programs that incorporate results of current research efforts on how to structure interprocess communication to admit the most parallelism. For example, Lamport [Lamport, 1985], Peterson [Peterson, 1983], and Vitanyi & Awerbuch [Vitanyi and Awerbuch, 1986] present algorithmic solutions for the concurrent-reading-while-writing (CRWW) problem that permit concurrency among readers and writers, as well as among writers themselves. Instrumentation to support execution replay can be added to systems that use such protocols, provided that a legal linear order of operations (consistent with their real-time ordering) on each shared object can be constructed.

With the existence of a valid linearization ensured, a series of modifications to a shared object can be represented as a totally-ordered sequence of versions. Each version has a corresponding version number, which is unique with respect to a particular object. During normal program execution (*i.e.*, *the monitoring phase*) we record a partial order of the accesses to each object based on these versions. (It is a partial order because the order in which multiple processes read a particular version of a shared object is unimportant.) To enable the partial order of operations to be ascertained, the current version number and the number of readers that have seen the current version are maintained for each object. The partial order is recorded during execution by having each process record the current version number of each shared object it accesses.[1] Each time a process performs an *update* operation that modifies the state of a shared object, it must also advance the version number and record the number of readers that saw the previous version of the object. During program replay, these traces are used to ensure that each process sees the same version of each shared object that it saw in the original execution. The relationship between the process interleavings possible under control of these traces and those in the original execution satisfies the *sufficiency* condition of Theorem 1; thus, processes see the same inputs and compute the same outputs, resulting in indistinguishable executions. As long as the recorded execution trace is available, the original program execution (or an indistinguishable one) can be produced over and over.

The goal of developing this execution tracing technique is to enable programmers to replay arbitrary executions of parallel programs. Since we cannot predict when it may be desirable to replay a particular execution, it must be practical for the monitoring mechanisms to be in place during every execution. Therefore, our mechanisms should have minimal impact on program performance. Since data-logging techniques record more data than our synchronization tracing technique for all but the finest grain process interactions, in nearly all cases our technique will record less information during an execution and have a smaller impact on program performance. For techniques that provide execution replay without recording data values communicated between processes, the *necessary* condition of Theorem 1 states that for a model in which processes perform read-modify-write operations on shared memory objects, we must ensure that the partial order of *reads* and *updates* with respect to each shared object is identical to guarantee our ability to replay indistinguishable executions. Since our communication model using shared objects similarly permits update operations (during exclusive access) whose effects depend on current object state, we must similarly guarantee the same partial order of operations on shared objects as during the original execution; thus, we must record the partial order of operations on each shared object during the original execution. Theorem 1 enables us to claim that for a synchronization tracing technique, the amount of information we record is **minimal** as no less information can guarantee an indistinguishable execution.

In the following sections, we illustrate our synchronization tracing technique using a CREW protocol for access to shared objects. A CREW protocol ensures a total order of writers (*i.e.*, *updates*) with respect to each shared object, a total order of readers with

[1] Each object version number is effectively a logical timestamp with respect to the object.

respect to writers of each shared object, and a partial order of readers with respect to each shared object. Since the execution path of a program can be characterized by a partial order on the operations with respect to each shared object, we will not require a total order.

A simpler variant of our synchronization tracing technique can be used to monitor protocols that ensure mutually-exclusive (ME) access to shared objects. A current version number is the only information that must be maintained on behalf of a shared object protected by a ME lock. Each process that acquires a ME lock need only record the current version number of the shared data object and advance the version number before the lock is released.

In addition to being independent of a particular protocol, our synchronization tracing technique does not rely on a particular granularity of interprocess communication. The granularity of access to shared objects is implementation dependent. Message-passing systems only require the protocol during shared buffer access; shared-memory systems may require the protocol to be used whenever shared storage is referenced. It is important to note, however, that the granularity of the objects monitored is inversely related to the relative overhead of monitoring and the volume of trace information that will be generated. Increasing the granularity of the objects monitored decreases the relative overhead associated with monitoring and reduces the volume of trace data. An important implementation decision is choosing an appropriate granularity for sharing data for which the costs of monitoring will be acceptable. Typically, when the granularity of shared objects is large enough that the overhead of locking (necessary to ensure consistency) is small enough to be acceptable, the additional overhead of synchronization tracing will also be acceptable.

4.2.2 Data Structures for Program Monitoring

Synchronization traces of the shared object accesses in a parallel program execution can be recorded either at the processes, or at the shared objects. On shared-memory multiprocessors, it is more efficient to record these traces at the processes to avoid the need for prolonged exclusive access to shared objects while recording synchronization information. However, synchronization tracing of shared object accesses in a distributed system might better be accomplished by recording synchronization information at the shared objects. With process-based trace recording in a distributed system, when a kernel on a remote processor deposits a message into a buffer on behalf of a process (an operation on the shared message buffer), the synchronization information (*e.g.*, object version number) for that buffer will have to be sent back to the process to be recorded; this end-to-end acknowledgement may otherwise be unnecessary when using an unreliable communication protocol in an unmonitored program. With object-based recording, the kernel merely has to append the identity of the sending process to a trace associated with the buffer and no end-to-end acknowledgement or extra messages are needed in the monitoring phase.

Here, we focus on the data structures for recording synchronization traces at processes on a shared-memory multiprocessor. To record the partial order of accesses to

objects that characterizes an execution, we use a set of *process history tapes*. Each process in an execution is assigned a corresponding process history tape. The two operations that a process can perform apply to a history tape are WriteHistoryTape, which is used during the monitoring phase to append a value to the end of the history tape, and ReadHistoryTape, which is used during the replay phase to read the next value from an existing history tape.

During the monitoring phase, upon creation, each process is assigned a history tape that is initially blank. For each read or write operation on a shared object by a process, the process records information about the version of the shared object it accessed on its private history tape. All history tapes created during the execution of a parallel program are linked together to form a tree. Each time a process spawns a child, a reference to the history tape of the child process is recorded on the history tape of the parent. This organization of history tapes enables each process history tape to be associated with the correct process during execution replay. Since each process independently computes and records its trace information as part of its object accesses, synchronization tracing in the monitoring phase does not restrict the parallelism available to the application.

During the replay phase, as each process is created it is associated with its existing history tape. As each process requests access to a shared object, information about how the access occurred in the original execution in the monitoring phase is read from the process's history tape and is used to guarantee that the process accesses the same version of the shared object during execution replay.

In addition to the information recorded on a process's history tape regarding interactions with shared objects and child processes, arbitrary details of a process's execution can be recorded on the tape for use during replay. Specifically, the resolution of certain interesting events can be recorded on the history tape in order to replay programs containing nondeterminism. The information recorded about such events can be used to re-create the same events during program replay. A mechanism to support the recording of these events would need to be added to the implementation of the programming language at the appropriate level (*i.e.*, compiler code generation or language run-time support). Such a mechanism would be appropriate to record the statement alternative chosen in a nondeterministic selection statement, whether or not a timeout interval had expired during execution, and clock values returned by system calls.

4.2.3 Access Protocols for Shared Objects

In order to properly record a partial order of the accesses to each shared object, a protocol that ensures a valid linearization is needed. In this section we describe such a protocol, a concurrent-read-exclusive-write (CREW) protocol that incorporates synchronization tracing hooks to support program replay.

The CREW access protocol consists of four procedures: entry and exit procedures for readers, and entry and exit procedures for writers. During the monitoring phase, these procedures enforce a CREW access protocol on shared objects and record a partial order of accesses to each shared object. During the replay phase, these same procedures are used to enforce the partial order recorded during the monitoring phase.

```
ReaderEntry (var object: shared_object_header);
    if mode = MONITOR then
        Exclusive_Lock(object.lock);
        AtomicAdd(object.activeReaders, 1);
        Exclusive_Unlock(object.lock);
        WriteHistoryTape(object.version);
    else
        // wait for version seen during monitoring phase
        key := ReadHistoryTape();
        while (object.version != key) do delay;
    end if;
end ReaderEntry;

ReaderExit (var object: shared_object_header);
    AtomicAdd(object.totalReaders, 1);
    AtomicAdd(object.activeReaders, -1); // ignored in replay mode
end ReaderExit;
```

Figure 4.1: A CREW Shared Object Access Protocol for Readers.

Each process that reads a shared object must use the entry procedure **ReaderEntry** shown in figure 4.1. This routine uses an exclusive lock associated with the object to ensure that readers are not granted access to the object while a writer is using it. Once the reader is granted access by the lock, it increments the number of active readers using the object.[2] Writers are not allowed to modify the object as long as the count of active readers is nonzero. Once the count of active readers has been updated, the reader process releases the lock and records the version of the object it is about to read on its process history tape. Then, the reader is allowed to access the object. Eventually, the exit routine **ReaderExit** (also in Figure 4.1) is called, which simply maintains a count of all readers for a particular version of the object and decrements the number of active readers for the object, thereby allowing writers a chance to proceed.

In replay mode, the entry procedure for readers proceeds as before, except that history tapes are not written, they are merely read and advanced as execution proceeds. Each reader process must wait until the version number for the target object is equal to the version number recorded on the reader's history tape. This ensures that the reader will see the correct version of the target object during replay. Once the reader has read the object, a count of readers for that version is incremented in the exit routine. This counter ensures that a writer will not create the next version of an object during replay

[2]We use atomic increment and decrement operations to maintain the reader counts for an object, thereby avoiding the need for additional synchronization between entering and exiting readers. Without such primitives, synchronization would need to be added to protect against conflicting updates to the count of total readers (by multiple exiting readers) and the count of active readers (by multiple exiting readers and an entering reader).

until all readers have finished with the current version.

Each process that modifies a shared object must use the entry procedure **WriterEntry** (Figure 4.2). In this routine, the writer uses the lock associated with

```
WriterEntry (var object: shared_object_header);
    if mode = MONITOR then
        Exclusive_Lock(object.lock);
        // Wait for all current readers to finish
        WriteHistoryTape(object.version);
        while (object.activeReaders != 0) do delay;
        WriteHistoryTape(object.totalReaders);
    else
        // Read version modified during monitoring phase
        key := ReadHistoryTape();
        while (object.version != key) do delay;
        // Read count of readers for previous version
        key := ReadHistoryTape();
        while (object.totalReaders < key) do delay;
    end if;
end WriterEntry;

WriterExit (var object: shared_object_header);
    object.totalReaders := 0;
    if mode = MONITOR then
        object.version += 1;
        Exclusive_Unlock(object.lock);
    else
        AtomicAdd(object.version, 1);
    end if;
end WriterExit;
```

Figure 4.2: A CREW Shared Object Access Protocol for Writers.

the object to gain exclusive access to the object. Once the exclusive lock is acquired, the writer process waits for all active readers to finish. No new readers can access the object since the entry routine for a reader must also acquire the lock. When all currently active readers have finished with the object, the writer is free to access the current version of the object. The writer records the current version number of the object onto its process history tape, as well as the number of readers for that version. The writer may then modify the shared object. Exclusive access is maintained because the lock is not released until the exit procedure is called. The **WriterExit** routine (also in Figure 4.2) simply initializes the number of readers for the new version, increments the version number for the object, and releases exclusive access to the object by relinquishing the lock.

In replay mode, the object lock is not required for either readers or writers because the information on process history tapes, in conjunction with the counts maintained with the object, is sufficient to correctly order the operations on a target object. A writer must wait until the current version of the object matches the version number recorded on the writer's history tape. This ensures that the writer modifies the correct version. Next, the writer must make sure that the number of readers that have seen the current version of the object during replay is equal to the number of readers that saw that version in the original execution. Since the **ReaderExit** routine updates the count of total readers for the object version after completing the read, a writer cannot proceed until all reads of the previous version have finished. Following the write operation, the **WriterExit** procedure simply initializes the number of readers for the new version and then increments the object version number. Since this is the last operation performed by a writer, no reader will attempt to access the new version until the writer has finished.

This description of a CREW access protocol is intended to be illustrative, not definitive. Our execution tracing technique requires neither a CREW protocol nor this particular implementation of a CREW protocol. (In fact, the **ReaderEntry** routine in the CREW protocol presented here has a small critical section protected by an exclusive lock. An alternate (more efficient, although slightly more complicated) CREW lock implementation that avoids using a critical section (which unnecessarily serializes readers) is given in Appendix A.) As stated previously, we could use an ME protocol to guarantee a valid linearization. A different implementation of a shared object locking protocol would be required in a loosely-coupled system, in particular, one that does not use shared memory. Also, as mentioned earlier, in a loosely-coupled system, it would likely be more efficient to record synchronization traces at the objects rather than at the processes, as presented in the protocol given in this section. Version numbers could be used to control access to message buffers on remote nodes, preventing buffer overflow problems during replay. However, in a loosely-coupled system, additional mechanism would be needed in the replay phase (possibly requiring extra "synchronization messages") to control access to shared communication buffers. Nevertheless, regardless of the characteristics of a particular implementation of the access protocols, our basic approach is to record a partial order of operations on each shared object and ensure the same order during program replay.

4.3 A Multiprocessor Prototype

A prototype implementation of our synchronization tracing technique has been developed for the BBN Butterfly™ Parallel Processor. Several considerations motivated the choice of the Butterfly as a testbed. First, we had a Butterfly at the University of Rochester, but lacked methods and tools for debugging parallel programs. This, combined with the surge of software development for the Butterfly, created an urgent need we wanted to fulfill. Second, interprocess communication on the Butterfly is inexpensive, which tends to encourage development of communication-intensive programs. Third, communication on the Butterfly is available over a wide range of granularities; process interactions can occur through direct sharing of memory, or through the use

of higher-level primitives for message-passing. Finally, the high degree of parallelism offered by the Butterfly provides a challenging test since highly parallel, communication-intensive applications will experience the greatest performance degradation using any program monitoring technique.

4.3.1 The BBN Butterfly Parallel Processor

The BBN Butterfly Parallel Processor at the University of Rochester consists of 128 processing nodes connected by a switching network. Each switch node in the switching network is a 4-input, 4-output crossbar switch with a bandwidth of 32 megabits/sec. Each processor is an 8 MHz MC68000 with 24 bit virtual addresses. A 2901-based bit-slice co-processor interprets every memory reference issued by the 68000 and is used to communicate with other nodes across the switching network. All the memory in the system resides on individual nodes, but any processor can address any memory through the switch. A remote memory reference (read) takes about 4 μs, roughly 5 times as long as a local reference.

Chrysalis [BBN Laboratories, 1987], the Butterfly operating system, consists largely of a protected subroutine library that implements operations on a set of primitive data types, including event blocks (structures used by processes to post a word of data to the event owner), dual queues (queues that hold a sequence of long word data enqueued by processes, or alternatively, a sequence of process handles corresponding to processes waiting to dequeue data as it becomes available), shared memory segments, and a global name table. Objects of these types can be shared among all processes executing on the machine. Low-level operations on these data types are provided by Chrysalis, many of which are implemented by microcode. These primitive operations provide a general framework upon which efficient high-level communication protocols and software systems can be built. Unfortunately, Chrysalis does not provide file system support. For this reason, our machine is not equipped with any secondary storage devices and is usable only as a back-end machine.

4.3.2 Monitoring Chrysalis Operations

Our prototype implementation provides programmers with encapsulated versions of the Chrysalis primitive operations on events, dual queues, shared memory objects, and processes. The encapsulated versions of the Chrysalis primitives incorporate instrumentation for synchronization tracing as detailed in section 4.2. This implementation was done at the level of primitive Chrysalis operations to make replay available to all programs; it can be used in any software system developed on top of the Chrysalis operating system. In particular, system development efforts at the University of Rochester that can be easily modified to incorporate our synchronization tracing technique include LYNX, a programming language and run-time system for distributed computing [Scott, 1986a; Scott, 1986b], and SMP, a message-passing system that supports multicast message communication among groups of processes [LeBlanc *et al.*, 1986; LeBlanc, 1988].

While encapsulating the Chrysalis primitives for events and dual queues, it became apparent that providing a CREW protocol for all operations was inappropriate. Most

of the operations on events and dual queues are atomic, which means that the operations must occur serially with respect to their target data object (a characteristic of the hardware). The CREW protocol allows concurrent readers of shared objects, but introduces additional cost. Since event and dual queue operations cannot exploit concurrent execution of readers, the expense of the CREW protocol is not justified. By replacing the CREW protocol with the simpler mutual exclusion (ME) protocol, we force serial execution of primitive operations on Chrysalis events and dual queues, but reduce execution overhead by simplifying the entry and exit protocols. An ME protocol enables use of a single entry/exit routine pair and reduces the amount of information recorded on process history tapes, since we need not maintain a count of the readers for each version.

Using encapsulated versions of Chrysalis primitives in program code requires no additional effort beyond that necessary to use the original primitives. Additional program code is only necessary for regulating access to shared memory objects. Chrysalis provides primitives for sharing segments of memory. General sharing of memory objects as provided by the Butterfly hardware and Chrysalis primitive operations impose no restrictions on memory access other than serializing word operations on each node, since the memory hardware has only a single port. To guarantee the consistency of data objects in these shared segments, programmers need to use access control primitives. We require that either programmers use our provided ME or CREW access entry and exit routines to control sharing in these segments, or incorporate the necessary synchronization tracing instrumentation into their own primitives. The programmer can control the granularity of operations bracketed by the access routines in response to performance concerns. By controlling the cost of the operations within an entry and exit routine pair, the programmer can balance the reduction of parallelism incurred when locking for long periods of time with the overhead of frequently executing the locking primitives.

4.3.3 Case Studies

Two applications were chosen for experiments in synchronization tracing and replay: computation of a knight's tour of a chess board and Gaussian elimination. The knight's tour problem was chosen because there is an existing implementation on the Butterfly that exhibits extremely nondeterministic behavior. A parallel implementation of Gaussian elimination was chosen for study since, unlike the knight's tour program, no matter what execution path occurs when the Gaussian elimination program is run, the overall amount of computation performed by the program is constant. Also, since this implementation of Gaussian elimination has already been the subject of a thorough performance study [LeBlanc, 1986], the statistics previously obtained about the program's execution behavior can be used as a baseline for comparison to determine the cost of our synchronization tracing techniques.

Knight's Tour

A knight's tour is a path on a chess board for a knight that successively visits each square once and only once using legal chess moves. Our program to compute a knight's

tour of a chess board consists of a master process and a user-specified number of slave processes. The master selects an initial position of the knight on the chess board and enters the corresponding board description in a global task queue. Next, the master creates a set of slave processes that cooperate to search for a knight's tour beginning with the initial board position. Each slave removes a set of board descriptions from the global task queue and replaces it with a new set of board descriptions that could be generated by adding a legal move of the knight from its previous position. The order that these board descriptions are added and deleted from the task queue determines the breadth and depth of the search performed. Since the order in which slave processes are granted access to the task queue depends on the relative progress of the processes and resolution of contention for the task queue, successive executions of the program tend to produce different tours.

Calls to monitored versions of the task queue primitive operations (the task queue is a dual queue) were inserted in the program in place of the original calls to Chrysalis primitives. These modifications required minimal effort and caused no significant growth in code size. The effect of the monitoring on the performance individual primitive operations on the task queue is substantial. Chrysalis dual queue primitives are implemented in microcode; however, they provide no support for obtaining a serialization order or maintaining a process history tape. Therefore, we must use a lock to control the ordering of primitive operations on the task queue and maintain the serialization order in software at additional cost. How this additional cost affects overall program performance is difficult to measure due to the inherent nondeterministic nature of the knight's tour computation. We cannot obtain identical executions of the monitored and unmonitored versions of the program to compare execution times because such times vary wildly between successive invocations of the program. We were able to measure accurately the comparative execution times for a knight's tour program during the monitoring phase and the replay phase of the same execution. The difference between the two execution times was less than 5%.

Using 16 processors, three successive executions required 18, 38, and 52 seconds to find three different solutions for a 5×5 chess board; the executions used 12K, 36K, and 60K bytes, respectively, for history tapes.[3] Using 64 processors, a solution was found in 43 seconds and required 48K bytes for history tapes. It is not surprising that the amount of space required for the history tapes of the knight's tour program varies with the amount of time taken to find a solution. Communication is roughly a constant percentage of the computation and no matter how many processors are working on the task, communication speed, hence history tape space requirements, is limited by the need to serialize access to a single shared task queue. We estimate that the knight's tour program accesses the task queue between 250 and 300 times per second; each access to the queue requires four bytes to record. From this we can estimate the space requirements for the history tape as a function of the time needed to find a particular solution.

[3]Our current implementation uses a 32 bit word for each entry on a history tape, although 16 bit words would suffice for our case studies, as well as most other programs. Therefore, our space requirements are conservative and could easily be reduced by a factor of 2.

Gaussian Elimination

To obtain an empirical comparison of the relative cost of monitored and unmonitored program executions, an existing program to solve a system of linear equations using Gaussian elimination was instrumented. In Gaussian elimination, the total amount of work performed by the program is independent of the precise ordering of interprocess events during execution; the computation for each pivot row depends on a fixed number of other rows.

The implementation of Gaussian elimination uses a broadcast message-passing system as the basis for communication among the cooperating processes in the program.[4] A single master process initializes shared data structures and then spawns worker processes to compute an upper triangulation of the matrix. The master delegates rows of the matrix to each slave process participating in the solution. Each time the processing of a row is completed, the contents are broadcast by the process holding that row to each of the other slaves.

To instrument this application we replaced some dual queue and event primitives used for synchronization between the master and slaves with monitored versions of the Chrysalis primitives. The underlying message-passing system, however, required more extensive changes. Message-passing was implemented using shared memory segments as communication buffers. Modifications to the send and receive primitives of the message-passing system were required to enforce the CREW access protocols, as detailed in section 4.2.3, for the shared communication buffers.

Although the code overhead and programming effort to make this transformation were more substantial than that required for the knight's tour, the size of the effort was still small. The original Gaussian elimination program contains 1059 lines of code. To instrument the program for execution replay, 24 lines of code were altered and 17 lines of code were added. Most of the changes to the source code files occurred in the message-passing module. Figure 4.3 shows the skeletal form of the monitored message-passing routines.

The performance of the Gaussian elimination implementation was degraded by the enforcement of a CREW protocol on shared object access and recording the access order to shared objects. Figure 4.4 depicts the performance of monitored and unmonitored versions of the application on a 400×400 matrix. The unmonitored program improves dramatically in performance as additional processors become involved in the computation, however, there is no significant improvement in performance when more than 32 processors are in use. In fact, performance begins to degrade slightly beyond 32 processors because the additional communication involved is not justified by the gain in parallelism [LeBlanc, 1986]. Our first attempt at monitoring this program recorded synchronization traces on history tapes for each shared object (rather than on history tapes for the processes as described in section 4.2.2) resulting in severe performance degradation for the Gaussian elimination program when more than 8 processors were

[4] The message-passing system used here is an early prototype of SMP [LeBlanc *et al.*, 1986; LeBlanc, 1988]. The results described in this section are particularly relevant to programs based on SMP, or similar communication models.

```
Send Message
    Find free buffer
    WriterEntry(buffer)
    Copy message into buffer
    Set number of recipients
    WriterExit(buffer)

Receive Message
    ReaderEntryPoll(buffers)
    Poll incoming message buffers
    Copy message into user area
    ReaderExitPoll(buffers)
    WriterEntry(buffer)
    Decrement number of recipients
    WriterExit(buffer)
```

Figure 4.3: Skeletal Message Passing Code Used by Gaussian Elimination.

in use, as shown in Figure 4.4. This degradation was due to the need for a critical section guaranteeing each process exclusive access to an object's history tape while it recorded synchronization information about an operation on the shared object. The performance degradation caused by this preliminary implementation of an access protocol with synchronization tracing demonstrated the need for a more efficient technique. Modifying the buffer access protocols to use process-based recording of synchronization traces reduced the need for critical sections and greatly improved the performance, but still managed to roughly triple the execution time of the program on 64 processors. Examination of the monitoring cost showed that the program was spending a great deal of time monitoring and recording noncritical polling operations on buffers. To lower the cost of monitoring, we devised a special entry procedure for use with the common programming idiom in which readers poll before reading a value.

Our implementation of message-passing uses polling to find incoming messages. Whenever a process attempts to receive a message, a large number of buffers, one for each process in the computation, are polled. Our naive approach to monitoring operations considered each polling operation as an access to a shared object, which was duly recorded on the process's history tape. The realization that none of the polling operations, *except the last one*, are necessary for replay led us to devise a special entry procedure used in conjunction with polling. With this new entry procedure, the access ordering to a buffer is recorded only when a message is found. An indication of which buffer supplied the message and the version number for that buffer are recorded on the process's history tape. During replay, only the buffer from which a process received a message during the monitoring phase is polled. Use of this entry procedure eliminated recording of nonessential ordering information during the monitoring phase, saving both time and storage space for the information collected. The performance of the program

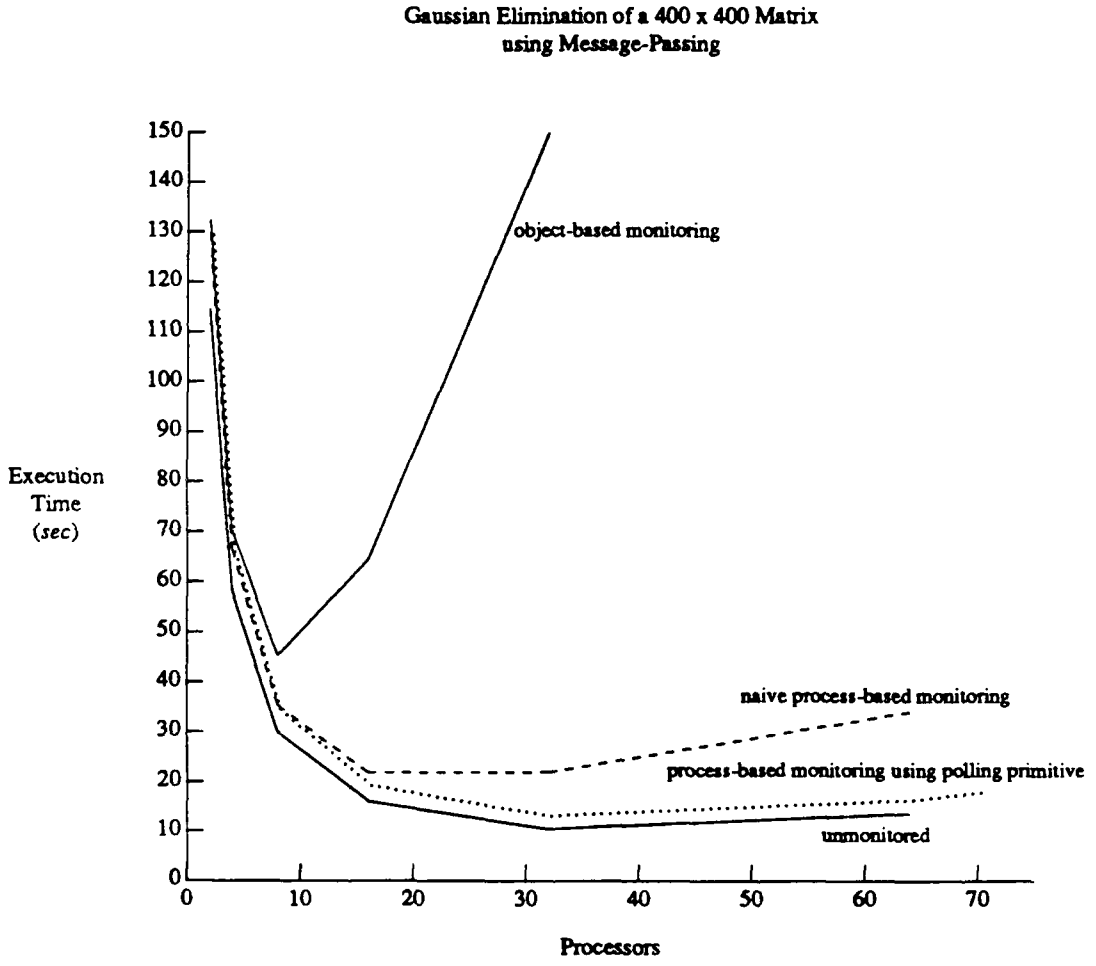

Figure 4.4: Synchronization Tracing Impact on Gaussian Elimination.

using the special entry procedure is also shown in Figure 4.4. The result: we were able to monitor a communication-intensive application for replay by imposing a performance overhead of less than 1% for up to 64 processors. In addition, we were able to replay the program in the same amount of time as was used by the execution in the monitoring phase.

As we have already stated, Gaussian elimination is a communication-intensive program, which tends to produce large history tapes. Diagonalization of an 800×800 matrix on 64 processors requires 400KB for the history tapes. While this is not a small amount of space, it is worth comparing the space requirements for our method with other techniques that save the contents of every message received by a process in an event log. Such an approach requires over 150MB of space! In general, synchronization tracing will always take less space than an event log whenever large messages are involved, since we only require between 4 and 8 bytes for each message.[5]

[5]Normally, four bytes per message are used, however, the polling entry procedure used by Gaussian elimination requires 8 bytes. The polling primitive must record an index indicating which buffer of the available set was accessed in addition to the buffer version number.

5 Coping with Asynchronous Events

Implicit in the synchronization tracing approach we present in the previous chapter is the assumption that a sequential process will always follow the same execution path when given the same input. This assumption is not true in the presence of an asynchronous transfer of control. Interrupts may occur at any time, causing a transfer of control to a completely different context. Asynchronous events due to out-of-band messages, timeouts, and hardware failures have the same effect. In all cases, the state of shared variables and global resources may be affected by the exact state in which event handler is asynchronously invoked. An inability to reproduce the asynchronous transfer of control at precisely the correct moment makes it nearly impossible to debug these types of programs using traditional cyclic debugging techniques. At a recent workshop on parallel and distributed debugging, this problem was noted by the developers of other debugging systems [Elshoff, 1988; Pan and Linton, 1988], none of whom had solved it.

This chapter presents a software technique for pinpointing the occurrence of asynchronous events during a program execution. This technique can be used to support virtual machine environment simulations for program executions affected by asynchronous events. We evaluate the overhead of using this technique on an individual process by measuring its impact on the executions of several sequential programs.

Pinpointing the occurrence of an asynchronous event in the execution of a process requires a measure of how far execution has progressed since the process began execution. For straight-line code, the value of the program counter adequately reflects the progress of an execution. However, for code with loops or procedure calls, the value of the program counter alone is inadequate. For such programs, a hardware instruction counter can provide a precise measure of an execution's progress by counting how many instructions a process has executed since creation. An instruction count can identify a unique point in the execution for a deterministic process since, if the process is presented with the same inputs, it will take the same execution path.

Although a hardware instruction counter has been recognized as useful architectural support for program debugging and profiling [Cargill and Locanthi, 1987], most processors do not yet provide even this basic facility (a notable exception is the HP Precision [Hewlett-Packard, 1987]). Thus, on most machines, pinpointing the occurrence of asynchronous events will have to be accomplished using software techniques. Here we

describe a software representation of an execution's progress that we term a *software instruction counter* (SIC).

There are several approaches that could be used to implement a SIC. The most obvious approach is to simulate a hardware instruction counter. That is, we could preface each instruction in a program with an instruction that increments a count in memory. This count will differ at most by one from the number of program instructions executed. Although an obvious implementation, it has an equally obvious drawback: both the code size and execution time would increase by a factor of two. A practical implementation must minimize both the number of instructions and memory references used to implement the instruction counter.

An alternative implementation might count only basic blocks. A software counter could be incremented upon entry to each basic block; the program counter would identify the precise location within a basic block. By augmenting a compiler to incorporate such a SIC in programs during compilation, we could (a) take advantage of the knowledge of basic blocks already present in the compiler, (b) allocate a register to hold the software counter, and (c) localize the required instrumentation within the compiler.

The cost of a software instruction counter can be further reduced. Basic blocks are a static representation of the computation. A basic block represents a set of consecutive instructions that must be executed in sequence, however consecutive basic blocks may or may not be executed in sequence. We do not need to increment the instruction counter if a basic block falls through to the next basic block; we need only count branches to the start of a block. In fact, only *backward* branches and subroutine calls need be counted, since we cannot reuse a particular program counter value without branching backwards. Thus, a combination of the program counter and a count of the number of backward branches taken during an execution are sufficient to uniquely identify any state in the computation.

In the next section we describe the implementation of a software instruction counter based on this idea.

5.1 Instrumenting Programs with a Software Instruction Counter

An instruction counter must be able to measure the progress of a program and interrupt the program after a certain amount of progress has been made [Cargill and Locanthi, 1987]. To incorporate a software instruction counter within a program, we must modify the program, and any library routines it calls, to update a counter on each backward branch or subroutine call, and to test (perhaps implicitly) the value of the counter each time it changes value. When the counter reaches some predetermined value, a transfer of control to an appropriate handler must be arranged.

This program modification can best be performed by a compiler, since the information available during compilation is precisely what is necessary to add the appropriate code. However, considerable effort is required to augment a production compiler to

instrument programs during compilation. Instead of modifying a compiler, for our prototype we direct the compiler to generate assembly code, and use a separate program to instrument the assembly code before passing the code to the assembler. Despite the decoupling of program instrumentation from compilation, this approach achieves results nearly as good as those we would expect from a compiler-based instrumentation system, without requiring significant compiler modifications.

As the basis for our experiments with instrumenting programs to incorporate a SIC, we use the GNU optimizing C compiler (*gcc* version 1.31). One factor that influenced this choice is that *gcc* supports a command-line option that can direct the compiler to leave any of the machine registers unused by the assembly code it produces. This option enables us to reserve a general-purpose register for counting branches. Without this capability, our SIC implementation would have to maintain a branch count in memory, at a considerable cost in performance. A second factor motivating the use of *gcc* is that it has a high quality optimizer. Optimization is important for accurately measuring the overhead of a software instruction counter; without optimization, overhead due to the SIC could be dominated by overhead due to poor code.

The second component of our instrumentation system is the *assembly code instrumentation program* (ACIP), which scans assembly code and instruments it to count backward jumps and subroutine calls. Since ACIP deals with assembly code, it is necessarily a machine-dependent part of our instrumentation system.

To simplify both ACIP and the code it produces, we take advantage of a simple observation: maintaining an exact count of the number of backward branches and subroutine calls is not necessary to make the SIC scheme work. During execution, it is permissible for the SIC to count both conditional branches that are not taken and forward branches. As long as the SIC increases monotonically and is consistently incremented *at least once* at each backward branch and subroutine call that occurs during program execution, a value of the PC will not be re-used without the value of the SIC changing. With this condition satisfied, each (PC,SIC) pair uniquely identifies a particular instruction in the execution history of a program.

ACIP uses two passes over its assembly code input. In the first pass, ACIP identifies labels in the code and inserts each label along with its statement number into a symbol table. The symbol table handles global, local, and numeric labels, making it possible to use ACIP to instrument hand-generated assembly code as well as compiler-generated code. In the second pass, ACIP examines each statement to determine if it is a branch or a subroutine entry point.[1] All statements that do not fall into these two categories are echoed to the output unchanged. If the statement is a subroutine entry point (*gcc* sets up a frame pointer for each subroutine making entry points easy to recognize), code to count the subroutine call is added. If the statement is a branch, ACIP makes a simplifying assumption: unless the target location of a branch is specified using a simple alpha-numeric label (not a PC-relative target, indirection through a register, or a target expression involving label arithmetic), the branch operation is assumed to be

[1] Changes in flow of control due to subroutine invocation can be counted at the point of call or inside the subroutine body. We chose to instrument each subroutine body since this strategy causes the least growth in code size.

backward and is instrumented accordingly. This assumption may cause unnecessary instrumentation of some branches in the target program; however, without modifying the compiler to perform the instrumentation task or requiring ACIP to assemble its assembly code input and understand the semantics of jump tables, this assumption is unavoidable. For each branch whose target is a simple alpha-numeric label, the symbol table is queried about the location of the target. Forward branches are not instrumented, but backward branches and branches to locations defined externally are.

In the following sections, we assume that a register can be dedicated to support a software instruction counter.

5.1.1 SIC on a CISC

Implementing a SIC requires maintaining a counter value and transferring control to a handler when that count reaches a predetermined value. Complex instruction set computer (CISC) processors, such as the VAX [Digital Equipment Corporation, 1981] and the Motorola 68020 [Motorola, 1985], generally supply loop control primitives (*e.g.*, decrement and branch) that can be used to efficiently implement a SIC. Sample code sequences to count backward branches and subroutine calls using these loop control instructions are shown for the 68020 and VAX in table 5.1. In each case, when the SIC register reaches -1, control transfers to the routine *SICrefOflw*, which handles underflow of the SIC register. Unconditional backward branches and branches to unknown target locations are instrumented similarly.

Sequence Type	Original Code Sequence	Sequence With SIC Instrumentation	
		68020	VAX
Cond Branch	L1: ⋮ < *compute cc* > beq L1	L1: ⋮ < *compute cc* > dbne L1,d7 bne 1$ jsr SICrefOflw 1$:	L1: ⋮ sobgeq r7,1$ jsr SICrefOflw 1$: < *compute cc* > beq L1
Subroutine Entry	Entry:	Entry: dbra 1$,d7 jsr SICrefOflw 1$:	Entry: sobgeq r7,1$ jsr SICrefOflw 1$:

Table 5.1: Sample Instruction Sequences for Implementing a SIC on a CISC Processor.

On the 68020, we use the decrement-and-conditional-branch (*dbcc*) instruction to maintain a branch count in a register. The *dbcc* instruction takes three parameters: a condition code, a data register Dn, and a branch displacement. The semantics of the 68020 *dbcc* instruction are given in figure 5.1. The 68020 *dbcc* instruction does not alter the condition codes.

```
if not cc then
 Dn ← Dn - 1;
 if Dn ≠ -1 then PC ← PC + disp fi
fi
```

Figure 5.1: Semantics of the 68020 *dbcc* Instruction.

On the VAX, we use the subtract-one-and-branch instruction (*sobgeq*) to maintain a branch count in a register. The *sobgeq* instruction takes two arguments: an index operand, and a displacement. The semantics of *sobgeq* are shown in figure 5.2. The

```
index ← index - 1;
if index ≥ 0 then PC ← PC + disp fi
```

Figure 5.2: Semantics of the VAX *sobgeq* Instruction.

VAX *sobgeq* instruction alters condition codes as part of its operation.

Although the loop control primitives on the 68020 and the VAX are similar, the 68020 *dbcc* is better suited to maintaining a SIC. As shown in table 5.1, the *dbcc* instruction folds the decrement of the SIC register in with the conditional branch. On the 68020, the *dbcc* costs the same number of cycles as a simple conditional branch, so SIC instrumentation introduces no overhead in the likely case where the branch is taken.[2] Since the VAX *sobgeq* instruction does not take a condition code as an argument, it cannot be used to directly replace a conditional branch. Therefore, the overhead of maintaining a SIC for a program on a VAX will be higher than on a 68020. Also, since the *sobgeq* instruction affects condition codes, use of it to maintain a SIC must either precede the computation of the condition codes for the branch, or be added at the branch target.

Without using these special loop control primitives on the VAX and 68020, the overhead of maintaining a SIC would increase. In both cases, it would be necessary to use the following instruction sequence to count a backward branch or subroutine call:

```
    dec register
    bgeq 1$
    jsr SICrefOflw
1$:
```

In the average case, when the register doesn't underflow, this instruction sequence adds an overhead of two instructions to count a backward branch or a subroutine call. However, with the loop control instructions, the average case requires at most one instruction to update the SIC register.

In our implementation for the 68020, ACIP replaces conditional branches with a *dbcc*. Since conditional branches may use a 32-bit displacement on the 68020, replacing

[2]Backward conditional branches are assumed to be taken more often than not, since most loops have more than one iteration.

them with a *dbcc* may introduce an assembler error since *dbcc* only allows a 16-bit displacement. C programs typically consist of many short functions, so it is unlikely that this assumption will cause problems. In practice, we have not encountered a program for which this assumption was violated.

5.1.2 SIC on a RISC

Reduced instruction set computer (RISC) processors support only simple instructions with the goal of having each instruction execute in a single cycle. Without complex primitives, such as the 68020's decrement-and-branch instruction (which enables SIC instrumentation of conditional branches with no overhead in the average case), counting each backward branch and subroutine call will cost at least one cycle to update the SIC register.

Surprisingly, examination of the instruction sets for RISC processors reveals that the overhead for maintaining a SIC is not uniform. The HP Precision RISC processor [Hewlett-Packard, 1987] provides a "recovery counter" that can be used as a hardware instruction counter. When the recovery counter is enabled, the execution of each non-nullified instruction causes a decrement of the counter; when the counter value goes negative, a trap is generated. The *SICrefOflw* routine can be installed as the handler for this trap condition. Processors that support addition or subtraction with trap on overflow such as the SPARC [Gardner, 1988; Muchnick, 1988], the MIPS R2000 [Moussouris *et al.*, 1986], the Am29000 [Am29000, 1988], and the MC88100 [MC88100, 1988], can update the count in the SIC register with a single instruction. Other processors that do not support arithmetic operations with trap on overflow (*e.g.*, the Fairchild Clipper [Fairchild Semiconductor Corporation, 1987]) require a multi-instruction sequence to update the count in the SIC register and check for overflow.

5.2 Cost Experiments

In this section, we describe a series of measurements and predictions of the execution overhead that SIC instrumentation would add to each of a set of sample programs for both RISC and CISC processors. First, we describe measurements of SIC overhead for a set of programs executing on the CISC 68020 processor. Then, using our measurements of overhead on the 68020, we derive some predictions of the overhead of adding a SIC to programs executing on the SPARC and MIPS R2000 processors.

In measuring the overhead of a SIC, two types of overhead are important:

1. direct overhead that results from executing additional instructions to maintain the SIC, and

2. indirect overhead that results from making a register unavailable for program use by dedicating it to a SIC.

To measure both the direct and indirect overhead, we compiled each test program three different ways using *gcc* with optimization enabled. For the *baseline* version of the program, the compiler was permitted to allocate all of the machine registers to the program. For the *register* version, the compiler was directed to reserve one of the general-purpose machine registers, making it unavailable for use by the program. In the *count* version, ACIP instruments the *register* version of the program to use the reserved register to maintain a SIC.

By comparing the execution times of the *baseline* and the *register* versions of the program, we can measure the indirect cost of taking a register away from each program for use by the SIC. By comparing the *baseline* and the *count* versions of the program, we can measure the total overhead for the SIC. The difference between the *register* and *count* versions measures the direct overhead. Table 5.2 shows the measurements of the overhead for several sample programs executing on a 68020.

	time in seconds			SIC
test program instance	baseline[a]	register[b]	count[c]	overhead
recursive Fibonacci	99.1	99.1	102.8	3.7%
compress	329.7	332.8	330.3	0.2%
grep '[a-z]+Z'	92.2	92.5	103.3	12%
grep 'ZZZ'	24.7	24.3	25.0	1%
ditroff	149.0	148.5	155.2	4.1%
lex	53.6	53.5	53.8	0%
Dhrystone 2.1[d]	108.4	108.4	122.0	12.5%

[a]All registers available (compiled with 'gcc -O').
[b]Register d7 unavailable (compiled with 'gcc -O -fixed-d7').
[c]SIC enabled (compiled with 'gcc -O -fixed-d7').
[d]Execution time for 500,000 Dhrystones.

Table 5.2: Measurement of Direct and Indirect Costs of SIC.

The impact of instrumentation overhead on the cost of subroutine calls is illustrated by measurement of the *Fibonacci* program shown in figure 5.3. Table 5.2 shows that the instrumentation overhead for the Fibonacci program is 3.7%.[3] The overhead due to maintaining a SIC is dwarfed by subroutine linkage, evaluation of the test, and computation of arguments for the recursive calls. We expect that typical subroutines have larger bodies than Fibonacci and therefore will likely incur less than 3.7% overhead for counting subroutine calls.

Execution timings for the compress program, a data compression utility, showed only a 0.2% overhead for maintaining a SIC. In comparing the *count* and *register* versions of

[3]After these measurements were taken, Neal Gafter pointed out that in a recursive call to a function, both the call and return are effectively backward branches in the control flow. Therefore, for a recursive function such as Fibbonacci, both the call and return transfers of control must be counted, effectively doubling the overhead.

```
main(){ fib(34); }
fib(i)
int i;
{
    if (i <= 1) return 1;
    return fib(i - 1) + fib(i - 2);
}
```

Figure 5.3: The Fibonacci Test Program.

this program, the *count* version, which maintains a SIC in a register, ran faster than the *register* version, which simply leaves a register unallocated. Possible explanations for this anomaly are that the addition of the SIC instrumentation to the compress code changed the page boundaries in the text segment resulting in a smaller working set, or that the new alignment of the instructions affected instruction pre-fetching or caching.

The SIC overhead of 12% for the string-matching program grep when presented with the regular expression '[a-z]+Z' is the highest measured overhead for a real program and thus requires some explanation. *Advance*, a short procedure, is the heart of the matching algorithm for grep. For this test case, *advance* is called once for each of the 4×10^6 characters in the test data set. *Advance* consists of a switch statement in which the cases encountered for this particular regular expression consist of only a few instructions. Since *gcc* produces code that uses indirection through a jump table to dispatch the switch statement, ACIP treats the branch to an indirect target as potentially backward and adds SIC instrumentation. Compiler-based instrumentation of this code would recognize that indirection through this jump table is used exclusively for forward branches and omit the SIC instrumentation. To measure the overhead contributed by the unnecessary instrumentation of the switch statement, the switch statement instrumentation was removed manually; a subsequent test with the same regular expression took only 97.7 seconds on average to execute. Without the unnecessary instrumentation of the switch statement, the SIC overhead is only 5.9%, which is comparable to the instrumentation overhead measured for the calls to short subroutines in the Fibonacci test. In both cases, the instrumentation overhead is magnified because the number of instructions in the subroutine call is not large enough to dominate the cost of counting the subroutine call. The second test for the grep program uses a pattern which does not require invocation of the *advance* procedure. The absolute time of this test is much shorter, even though both tests use the same test data set. Note that for this case, the SIC overhead is only 1%.

Most of the 4.1% run-time overhead for the ditroff test is likely due to the cost of counting subroutine calls. An execution profile of the ditroff program (generated using *gprof* [Graham *et al.*, 1982]) shows the execution time apportioned among a large number of calls to very short procedures. Also, in the ditroff test and the second grep test, the execution time of the program decreased when the compiler was given one less register to allocate to the program. Presumably, the compiler made a bad decision to

keep an infrequently used value in a register, where the overhead of saving and restoring the register across subroutine calls outweighed the benefit of faster access to the value.

The final program tested was the Dhrystone 2.1 benchmark [Weicker, 1988], which measures the integer performance of a compiler and machine pair. The execution overhead of adding a SIC to the Dhrystone was higher than all of the other programs measured. Most of this overhead is due to the very short procedures in the Dhrystone. We expect lower overhead for real programs since most programmers would use in-line procedures or macros for functions as short as those in the Dhrystone. Nonetheless, the results for the Dhrystone benchmark are useful because Dhrystone results are available for a wide range of machines. By determining the number of branches and subroutine calls counted in a single iteration through the Dhrystone, we can predict SIC overhead on a RISC using the Dhrystone figures reported for RISC processors.

For all sample programs, the indirect costs associated with sacrificing a register for branch counting were insignificant. Unless a compiler uses interprocedural register allocation, it typically will not utilize all of the registers inside procedure bodies; therefore, we expect that the indirect overhead for dedicating a register to an SIC will be small for most compilers. One shortcoming of the measurements that we performed was that we did not measure any programs using simulated floating point operations which typically are written in assembly code using all of the registers. In this case, the impact of allocating a register for the SIC would be greater. Even so, we found that in the Unix math subroutine library, only infrequently were all registers in use. This leads us to believe that the routines could be rewritten efficiently using one less register, even on a machine such as the 68020 which has only 8 general-purpose registers available.

Although we do not have direct measurements of the performance overhead of a SIC on a RISC, we can use published Dhrystone performance measurements of several RISC processors, coupled with our SIC measurements on a CISC, to develop performance projections for RISC architectures. Our method for predicting SIC overhead on a RISC is as follows:

1. We ran a Dhrystone benchmark on our 68020-based workstation for several different iteration counts. By noting the number of branches and subroutine calls counted for each test run of the benchmark program, the number of branches due to miscellaneous work other than the body of the benchmark was factored out and the number of branches for each execution of the Dhrystone was determined. We discovered that each Dhrystone executes 72 backward branches and subroutine calls that require update of the SIC.

2. For most RISC processors (*i.e.*, those that support integer arithmetic operations with trap on overflow), the update to the SIC register needed at each backward branch or subroutine call requires a single instruction. Thus, for these processors, an additional instruction would need to be executed for each branch in a Dhrystone. Since an instruction on a RISC takes one cycle, the the total cost of updating the SIC register in each iteration of the Dhrystone would be 72 cycles.

3. Using the Dhrystone results and the cycle times listed for machines with RISC processors in the December 1988 Usenet distribution [Richardson, 1988], the frac-

tional overhead for SIC instrumentation of the Dhrystone on these architectures was estimated by comparing the execution time of 72 RISC instruction cycles for SIC instrumentation to the total cost of executing a single Dhrystone (which is computable from the Dhrystone benchmarks).

The derivation of our equation for the expected overhead of maintaining a SIC on a RISC processor is shown below.

$$overhead\ (\%) = \frac{time\ spent\ maintaining\ SIC}{single\ Dhrystone\ execution\ time}$$

$$overhead\ (\%) = \frac{\frac{updates}{Dhrystone} \times \frac{instr.}{update} \times \frac{cycles}{instr.} \times \frac{sec}{cycle}}{\frac{1}{Dhrystones\ per\ second}}$$

Replacing $\frac{updates}{Dhrystone}$ by 72 (measured value), $\frac{instr.}{update}$ by 1, and $\frac{cycles}{instr.}$ by 1, and simplifying we get:

$$overhead\ (\%) = \frac{72 \times Dhrystones\ per\ second}{clockrate\ (Hz)}$$

Using this equation, we computed overhead predictions for maintaining a SIC on three RISC computers for which Dhrystone information was available; table 5.3 summarizes these results. All of the Dhrystone numbers shown in table 5.3 reflect compilation with the highest level of optimization (-O3 flag) for each compiler. Where several Dhrystone measurements for the same computer were available, we chose the highest value since this would give the most pessimistic estimate of the SIC overhead. The predictions presented in table 5.3 show that the overhead for maintaining a SIC for the Dhrystone benchmark on a RISC is comparable to the overhead measured for the 68020. Extrapolating from this prediction, it appears that the overhead for instrumenting programs with a SIC on RISC computers will be comparable to that on CISC computers. Our measurements on the 68020 and these predictions indicate that typical programs can be instrumented with a SIC with less than 10% overhead for either RISC or CISC architectures.

System	Processor	Dhrystones	Clockrate	Predicted Overhead
Sun 4/260	SPARC	18048	16.67 MHz	7.8%
MIPS M/500	R2000	12806	8.00 MHz	11.5%
MIPS M/1000	R2000	22590	16.00 MHz	10.2%

Table 5.3: Overhead Predictions for Branch Counting on a RISC.

5.3 Debugging with a Software Instruction Counter

To reproduce the effect of an asynchronous transfer of control, it is necessary to reproduce the transfer at precisely the same state in the computation. The program counter

is not a sufficient indication of the state of a computation, since it describes a static location in the code segment, not a dynamic location in the execution path. The real-time clock is also insufficient, since it usually lacks the necessary resolution. A hardware instruction counter would be sufficient, but is not strictly necessary. Instead, a combination of program counter and software instruction counter can be used to describe exactly the state of a computation when an asynchronous transfer takes place.

To replay such programs, our replay mechanism must cause the transfer of control to occur in the state in successive executions. We can use the SIC to represent the state in which the transfer occurred. To record an asynchronous transfer of control in an execution history, we instrument all interrupt and exception handlers in the program so that they record the value of the program counter and the SIC at the time of the transfer, as well as an indication of the type of trap or interrupt. (Since we are dealing with shared memory programs, our monitoring is *intrusive* in that the programmer must insert the appropriate code in the program.) The additional overhead is approximately eight instructions per handler.

During program replay, the execution history is used to guide execution. If replay is enabled, the record of the next asynchronous transfer is read from the execution history, which contains the value of the SIC at the time of the original transfer. The SIC register is initialized to this value, causing *SICrefOflw* to be invoked when the correct number of backward branches has occurred. At that time, a breakpoint is set at the location specified by the program counter value in the execution history record for the asynchronous transfer. Note that we do not require repeated executions of the breakpoint; we set the breakpoint only after we are certain that the next execution of the instruction of interest is the point of the transfer. When the breakpoint occurs, we synthesize the trap or interrupt using the information stored in the history. Once a transfer has been synthesized, we reset the value in the SIC register to the time of the next asynchronous transfer and repeat the process.

Besides its usefulness for accurately pinpointing the occurrence of asynchronous events during a program execution enabling them to be recreated at the same point in a program execution replay, a SIC can also be used to implement valuable debugging functions for a single process. In particular, a SIC can be used to implement watchpoints and reverse execution, without the hardware support previously thought to be required [Cargill and Locanthi, 1987]. We can implement watchpoints by having a SIC serve the same function as the hardware instruction counter in [Cargill and Locanthi, 1987]. The program execution is divided into intervals based on the value of the SIC. At the start of each interval the condition associated with the watchpoint is evaluated. Whenever the condition is met, the condition is known to occur during the previous interval. By dividing that interval into sub-intervals and restarting the process (possibly using checkpoints to avoid re-executing the whole program), the SIC could be used to isolate the basic block where the condition is first satisfied. Single-stepping through this interval would provide the exact instruction responsible for the condition. A similar approach, based on re-executing previous intervals, could be used to implement reverse execution.

6 Debugging and Analysis with Synchronization Traces

This chapter explores techniques for debugging and analysis of parallel program executions based on synchronization traces. In the first section, we explore the utility of execution replay based on synchronization traces. In the second section, we describe how these traces can be augmented with additional information to increase their usefulness for debugging and analysis. In the third section, we describe an integrated toolkit that enables these augmented traces to be effectively exploited.

6.1 Execution Replay in the Debugging Cycle

Program replay makes it possible to repeat indistinguishable executions of a parallel program as often as desired. Unfortunately, this capability does not automatically debug programs, parallel or otherwise. How then do we use the ability to replay indistinguishable executions to debug parallel programs? In this section we describe several techniques for error isolation that can be used together with our approach.

Our synchronization tracing technique makes it possible to reproduce indistinguishable executions of a parallel program as often as desired. Any behaviors that may have been ignored during previous observations can always be reproduced on demand for closer examination. This capability is especially useful analyzing output of parallel programs since (a) multiple processes tend to generate a lot of output, making it easy to miss important results and (b) programming environments for parallel architectures are not as mature as programming environments for sequential machines, and often lack tools for collecting and analyzing output data. However, the most important reason for reproducible behavior is that it makes cyclic debugging possible.

The simplest form of cyclic debugging is to add output statements to an erroneous program that provide additional details about the execution of the program. Successive executions can be used to provide successively greater detail about those parts of the program under suspicion. Generally, this technique is not effective with parallel programs because the addition of output statements can change the relative timing of operations within the program yielding a different execution sequence. With execution replay, however, any number of output statements can be added to the program without changing the execution sequence provided by the replay mechanism. In fact, *any type of statement may be added to the program during replay*, as long as the additions

do not affect the sequence of interactions with shared objects by each process. Thus, a programmer can debug parallel programs by adopting the same cyclic methodology for error isolation used for debugging sequential programs. We have found that this capability alone is a valuable tool for debugging parallel programs, particularly in the absence of other debugging tools.

Repeatable execution also makes top-down, interactive debugging possible. Hierarchical abstraction of detail is necessary to cope with the complexity of large software systems. Abstraction is particularly important in understanding the behavior of parallel programs. The programmer should not have to be concerned with the low-level details of execution of a parallel program, such as the interleaving of primitive operations. Instead, we are interested in the salient features of the execution that characterize its behavior. Our approach allows the programmer to start with a high-level view of a program's behavior, produced by normal output statements or an event mechanism such as Behavioral Abstraction.[1] By carefully refining that viewpoint, based on the information made available during each successive replay, the programmer can study erroneous behavior at any level of detail desired. As a result, one can diagnose program errors in a top-down fashion without wading through voluminous, irrelevant detail at each step.

Another common technique used to debug sequential programs is breakpoint insertion. Breakpoints are added to the program at interesting points in the code. Execution is suspended at each breakpoint, allowing the programmer to examine the system state. Breakpoints only suspend a single thread of execution, however, which is not sufficient for parallel programs consisting of multiple threads of execution. Inserting a breakpoint in one process of a parallel program will have an effect on every process that communicates, directly or indirectly, with the suspended process. In particular, breakpoints can change the relative order of events during execution, producing a different execution sequence each time. Fortunately, we can provide reproducible execution *even in the presence of breakpoints.* No matter how many breakpoints are encountered during replay, we continue to order operations based on the contents of history tapes. A process that is suspended by a breakpoint will eventually cause all other processes to wait for some shared object to be read or written (assuming a connected graph of process interactions). When the suspended process is allowed to continue beyond the breakpoint, it will eventually catch up to the other processes and the entire program will continue executing. Thus, it is possible to cycle through breakpoints in many different processes during program replay, examining system state for a different process at each breakpoint.

This use of breakpoints also allows the programmer to examine the global state of the computation. Due to communication delays and a reliance on local viewpoints, it is impossible to take an instantaneous snapshot of global state. However, all we really need to see are *meaningful global states* [Chandy and Lamport, 1985], consistent states based on the *happened before* ordering of Lamport [Lamport, 1978]. For example, if we

[1] In fact, execution replay provides a foundation for application of any technique for debugging or abstraction of program behavior. Any dynamic technique described in chapter 2 could be used to analyze an execution replay.

suspend a process P at breakpoint X, all events that occurred before P reached X should be reflected eventually in all other processes. In addition, other processes should not be allowed to proceed beyond any point that requires process P to proceed beyond X. This is a natural view of a computation since, if all processes are stopped as the result of setting a single breakpoint, the *happened before* relation cannot distinguish between the global state represented by all suspended processes and an omniscient snapshot of the global state during normal execution.

We can use breakpoints, in conjunction with execution replay, to provide the ability to halt parallel programs in a consistent state, as in [Miller and Choi, 1986], without the need for additional mechanisms. By setting a local breakpoint during replay we are, in effect, setting a breakpoint in the global state. When the local breakpoint is reached, we can see the exact state of the local process containing the breakpoint, and the exact state of all other processes as they block due to enforcement of the *happened before* relation. Differences between the state of each process in an instantaneous snapshot and what we see at a breakpoint reflect the natural degree of asynchrony between processes in the program.

A consequence of our breakpoint capability is the ability to support single-step execution of processes. Single-step execution can be used during debugging to trace the state transitions of an individual process or the effects of interprocess communication on the internal states of communication partners. We can replay a process using single-step execution because enforcement of the *happened before* relation ensures that asynchrony between processes remains within allowable bounds.

Synchronization tracing can also be used in conjunction with an event logging technique to enable repeatable execution of a subset of processes involved in a computation. As we have described it, our approach requires that the input to each process be recomputed during replay, rather than retrieved from an event log. This is both an advantage and a disadvantage. While our technique requires less time and space during the monitoring phase, it also requires that all processes be re-executed during replay. Global replay is a disadvantage if the computational requirements to replay a program are very large, particularly when it is unnecessary to recreate the entire original set of processes to isolate an error. By using an event log together with synchronization traces, we can re-execute the subset of processes in which we are interested and simulate the rest.

There is a tradeoff between the expense of maintaining an event log during normal execution and the expense of re-executing all processes during replay. The event log approach and synchronization traces represent two extremes, wherein the expense is shifted from the monitoring phase to the replay phase. However, a compromise between our technique and the event log approach is possible. When frequent replay of a subset of processes in a computation is desired, as would be the case when using cyclic debugging to isolate errors, it is possible to collect additional information in an event log during replay that would eliminate the need for re-execution of the entire program during subsequent replay. We can record in an event log all external inputs to the subset of processes of interest. This record would include both inputs from the external environment and inputs from processes not under scrutiny. Interactions involving processes to be re-executed during replay are recorded, as before, as partial orders on history tapes.

On subsequent executions, only the designated subset of processes would be re-executed and their interface with the external environment, including the other processes, would be simulated using the event log. Since we assume that the debugging methodology is cyclic, the set of processes that are simulated by an event log will grow larger as we look at fewer processes in greater detail (*i.e.*, top-down debugging). Note however that we would continue to use minimal execution tracing in the monitoring phase because it has the least impact on normal program execution and can be used to generate event logs during the debugging cycle.[2]

6.2 Augmenting Traces for Debugging and Analysis

The synchronization tracing technique proposed in chapter 4 records only the synchronization information necessary for execution replay with no identifying marks. In the previous section, we described the utility of these traces for cyclic debugging of executions of parallel programs. For this use, no identifying marks in the trace information are necessary since each trace entry is interpreted in the context an execution state that is indistinguishable from the one in which it was generated. However, by themselves these traces are useless since the information in them is indecipherable. In this section, we describe how to augment the these traces to increase their utility for debugging and enable a variety of performance analyses.

For a trace entry to be meaningful outside the context of an execution state indistinguishable from the one in which it was generated, it must be marked with an operation type code. Prefixing the trace information (consisting of an item or sequence of items) for each operation on a process history tape with a type code makes it possible to parse the contents of the history tape since each type code marks the start of a fixed-length record. The ability to parse the contents of process history tapes enables analysis of the dynamic use of synchronization and communication operations, including operation frequency and sequencing.

If we additionally annotate each operation trace entry with the identifier of the shared object that is the target of the operation, in a post-mortem analysis phase we can construct *execution history graphs* which represent process interactions through shared objects. A program execution is represented naturally as a directed acyclic graph (DAG) of process interactions. Each node in the graph corresponds to an occurrence of an operation on a shared object. Operations by a single process are linked by directed arcs which denote their temporal precedence (the sequence of trace entries on a process history tape corresponds to the sequence of operations performed by the process). Operations by different processes on the same shared object are linked by directed arcs that reflect operation precedence as inferred from the object version numbers which serve as a logical timebase for each object. These execution history graphs can be displayed as "space-time" diagrams modeled after those introduced by Lamport [Lamport, 1978] for processes which communicate using message passing. As in Lamport's space-time

[2] In extraordinary circumstances where even a single replay is impractical, process history tapes and a partial event log could both be recorded during the monitoring phase.

diagrams, we position the sequence of operations performed by each process along a line parallel to the "time" axis, and differentiate between processes by separating them in the "space" dimension.

A space-time diagram of a program execution can be a useful aid for debugging. In chapter 1 we described the difficulty of correctly implementing a communication strategy for coordinating the multiple processes in a parallel program; incorrect implementation of such strategies is a common source of error in parallel programs. Trying to infer the complex dynamic relationship between a set of processes during a program execution is very difficult using traditional state-based debugging tools to examine individual processes. With such tools, it is difficult detect and diagnose implementation errors that cause the actual communication pattern to differ with the programmer's mental model. Diagnosis of such errors with state-based tools often requires repeated, painstaking examination of the processes in the erroneous execution (such repeated examination is not necessarily possible without our techniques for execution replay). In contrast to the local views of a parallel program execution provided by state-based examination, space-time diagrams provide a global view of the communication patterns that occur during a parallel program execution. This global view often makes errors in a communication strategy readily apparent, especially for programs with large-scale parallelism. Communication in such programs tends to be highly regular and patterns are readily recognizable. This regularity results from the use of SPMD parallel programs in which each process executes identical code using a "virtual process id" to determine the unique aspects of its behavior (such as which other processes in the computation it should communicate with). Using space-time diagrams, differences between the actual pattern of communication in a program execution and the programmer's mental model are immediately noticeable. In chapter 7 we present a number of space-time diagrams that illustrate the regular structure of interprocess communication in programs that exploit large-scale parallelism and present an example that demonstrates how a space-time diagram can facilitate debugging.

Although the space-time diagrams we have described thus far are extremely useful for understanding the logical relationships between processes in an execution, they lack detailed information about temporal relationships. We address this problem by further augmenting the operation trace entries with timestamps obtained from the real-time clock on the node which the process is executing. (See appendix B for a CREW protocol that records these fully augmented traces.) Several aspects of how processes operate on shared objects during a program execution are of interest from a performance perspective. First, how long is the interval between shared object accesses? Second, how long is the interval between the time a process requests access to a particular shared object (by invoking an access protocol) and the time when access is subsequently granted? Third, how long does the process continue to hold the lock on the shared object before releasing it? By augmenting each synchronization protocol to record real-time clock values for "access requested", "access granted" and "access released", we can answer these questions among others.

Annotated with this timing information, our execution history DAGs (the underlying representation of our space-time diagrams) with weighted arcs are similar in form

to Miller's *program activity graphs* that represent interprocess events and the elapsed time between related events [Miller, 1985b; Miller, 1985a]. We represent temporal information in a space-time diagram by appropriately dilating or contracting it along the "time" axis. Long intervals of time appear as long nodes, or long arcs. While the time associated with intraprocess arcs is directly reflected by their lengths (since these arcs are parallel to the time axis), the temporal delay associated with interprocess arcs must be measured by the length of their projection on the time axis.

Our view of an execution is fundamentally different from the view provided by animation systems, such as Jade's Mona console [Joyce *et al.*, 1987] or Belvedere [Hough and Cuny, 1987]. In animation systems, a program is represented by a static structure, usually a regular communication structure, on which interesting dynamic events are superimposed over time. Temporal relationships are difficult to analyze using animation because they are presented temporally. The granularity of temporal relationships shown in a single frame of animation is limited because two events at the same location cannot be shown simultaneously. Since parallel program analysis in general, and performance analysis in particular, requires extensive analysis of temporal relationships, we make those relationships explicit using a spatial dimension in the presentation of an execution. Our approach provides an abstract view of an entire execution, as opposed to the abstract view of a single state of an execution provided by animation, which makes it possible to survey at a glance the communication patterns of a computation.

We use our execution history graphs and their graphical representation as space-time diagrams as a basis for performance analysis of program executions. Our execution history graph representation enables analyses of the dynamic relationships between processes. Such analyses include (but are not limited to) computing a critical path for the program execution, the ratio of communication to computation (which provides a lower bound on program speedup), the effective parallelism in an execution, and dynamic traces of the communication and synchronization behavior.

It is also possible to use execution history graphs to examine the sensitivity of the program execution to changing conditions. In particular it is possible to artificially vary the delay associated with communication to examine the effect of changing communication costs on overall program performance. It is important to note that performance results derived from such an exercise are estimates, since this approach assumes that the program will continue to follow the same execution path in the presence of varying performance parameters. Under actual conditions, a program execution could be dramatically different following such a parameter adjustment, since any such adjustment can affect the set of probable process interleavings. However, it is still possible to learn a great deal about parallel programs using such a technique, particularly when using it with programs whose executions are less sensitive to race conditions.

6.3 An Integrated Toolkit for Dynamic Analysis

In this section we describe the design of an integrated toolkit, based on the augmented traces described in the previous section, that facilitates top-down debugging and performance analysis of large-scale parallel program executions. The toolkit consists of facili-

ties for recording execution histories, a common user interface for interactive, graphical manipulation of those histories, and tools for examining and manipulating program state during replay of a previously recorded execution.

The foundation of the toolkit is a library of instrumented synchronization primitives that efficiently gathers synchronization traces during a program execution and uses stored traces to replay indistinguishable executions. Each invocation of a library primitive adds a trace entry (complete with operation type identifier, shared object id, and timestamps) to the history tape of the invoking process.[3] (See appendix B for a CREW protocol that records these fully augmented traces.) To support analysis of programs with large-scale parallelism and non-trivial execution time, these traces must be spooled to secondary storage devices rather than gathered in primary memory. Furthermore, the library must arrange for these traces to be recorded to secondary storage even in the event of abnormal termination of the program being monitored.

As described in the previous section, we use the process traces of an execution to build an execution history graph which represents the execution behavior of the program. An execution history graph can be built efficiently from the traces on individual process history tapes. As each process history tape is read, allocate a graph node for each operation, add an arc to the previous operation by the process (if any), and add a pointer to the new graph node to the list of operations associated with the operation's target shared object. When all traces have been processed, sort the list of operations associated with each object using their the object version number as the sort key. With a sorted list of operations for each object, interprocess arcs can be added efficiently to the graph during a linear scan of each of the lists. These execution history graphs serve as the primary representation of the program execution for debugging and analysis.

The central component of the integrated toolkit is a user interface which facilitates interactive analysis of execution history graphs, and interactive control and examination of executions during replay from stored traces. To support these activities, the user interface for the toolkit provides an interactive, graphical browser and a programmable command interpreter.

In our methodology, program analysis begins with a graphical view of the entire program execution in the form of space-time diagrams. We emphasize graphical views of executions because they make the communication structure obvious, whereas textual views often obscure the basic structure of communication. Other views, in particular textual data, are supported but the graphical view is assumed to be the most frequently used view. The graphical interface supports a zoom-in and zoom-out capability, so that the focus of interest can be very high level or limited to a single event. Top-down analysis is supported by this approach, in that every analysis begins with a high-level view of an entire computation, and proceeds to local views of events of interest. Detailed data regarding individual events is always available upon demand.

[3]It is important to note that when generating augmented traces, the instrumented synchronization primitives should append an operation's type code, shared object id, and access request time to the history tape of the invoking process before causing the process wait for the shared object to become accessible. Otherwise, if this information is recorded only after access has been granted, no trace information will be available to diagnose a deadlock situation in which access is never granted.

A second use for the graphical browser is to provide a high-level interface for examining and controlling execution replay. Using the browser, operations in an execution history can be marked to set an "event breakpoint" during a subsequent execution replay. This breakpoint capability differs from that afforded by traditional debuggers in that it corresponds to a breakpoint at a particular point in the execution of a process, rather than a breakpoint that occurs the first time a particular statement is executed. These breakpoints can be implemented in a manner similar to traditional conditional breakpoints. First, a breakpoint instruction is inserted in the code for the marked type of operation. During program replay, each instrumented synchronization primitive increments an event counter for the process performing the synchronization operation. Every time the marked process invokes an operation of the marked type, the breakpoint is encountered and execution is resumed unless the event count matches the index of the operation (the index of an operation is its place in the sequence of events in a process history) marked with the event breakpoint. The graphical browser can also be used to follow the progress of a program during an execution replay. For each process, a token could be advanced through the sequence of the process's operations in the graph as operations are performed.

Although many interesting analyses are possible based on a graphical view of an execution, the sheer size of an execution history graph makes it impractical to base all analyses on manual manipulation of the graph. For this reason, we provide a programmable command interpreter to examine and manipulate execution history graphs and to produce meaningful synoptic analyses. The set of possible analyses is extensible because the interface is programmable. Additional views of a program execution, such as the relational database view of PIE [Segall and Rudolph, 1985], can be programmed using this representation. Application-specific analyses can be programmed by individual users. The combination of a comprehensive, fine-grain representation of program executions and a general, extensible user interface results in a very powerful base for parallel program analysis.

The programmable command interpreter also supports commands for interactive, symbolic, state-based examination and control of processes during an execution replay (by a straightforward adaptation of a symbolic debugger for sequential programs).

6.3.1 A Toolkit Prototype

Figure 6.1 is an overview of the organization of our prototype integrated toolkit. In our implementation, the toolkit is distributed among three distinct, physical locations. The components shown on the right side of the diagram execute on the target multiprocessor. The components shown on the left side execute on the programmer's workstation. In the center of the figure are shared data resources that reside on a network file service. This partitioning of components was necessary since our Butterfly is a back-end machine without any secondary storage. An ideal configuration would be to run all of the components on the parallel machine and use the capabilities of the X Windows System [Scheifler and Gettys, 1986]. to support use of the user interface from a remote workstation.

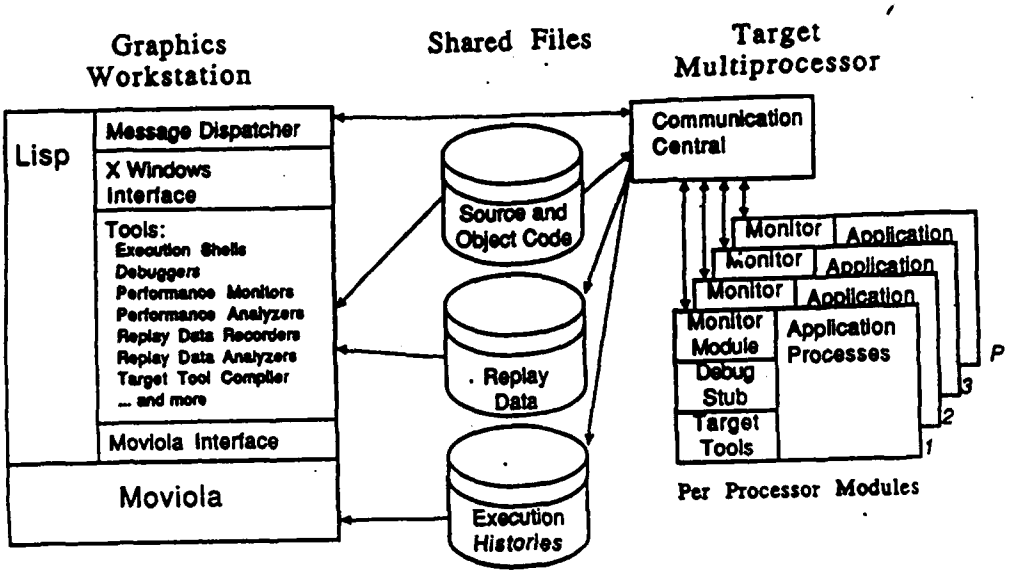

Figure 6.1: Organization of the Prototype Integrated Toolkit.

The shared data resources include source code, object files, execution histories, and replay statistics. The source code includes both the application program and libraries of instrumented synchronization primitives necessary to record the histories. The object file is used to replay an execution; the source code is used for symbolic debugging during replay. Execution histories are recorded both during the original execution of a program, and whenever the program is re-executed without the constraints of an execution history. Replay statistics are additional execution information recorded during program replay to support more detailed execution analysis than is possible with only the the minimal synchronization and timing information recorded during the original execution phase.

Workstation Components

The user interface for the toolkit resides on the programmer's workstation and consists of two major components: an interactive, graphical browser for analyzing execution histories, and a programmable Lisp environment. Communication among the workstation's components of the toolkit is routed through a dispatcher. This interface between the Lisp world and the rest of the system uses a simple message-passing model.

The execution history browser, called *Moviola*, is written in C and runs under the X Windows System. Moviola provides a graphical view of execution histories based on the DAG representation of processes and synchronization. Moviola gathers traces from individual process history tapes and combines them into a single, global execution history in which each edge represents a temporal relation between two events.

Figures 7.2 through 7.10 in chapter 7 show execution history diagrams created by Moviola. In each diagram, time flows from top to bottom; all edges in the DAG are implicitly directed from top to bottom. Events that occur within a process are aligned vertically, forming a time-line for that process. Edges joining events in different processes reflect temporal relationships resulting from synchronization. Event placement is determined by global logical time computed from the partial order of events collected during execution. Each event is displayed as a shaded box whose height is proportional to the duration of the event. Since events correspond to synchronization primitives, their duration represents waiting time. Events entailing little waiting have very small height and appear as horizontal lines.

Two parameters in the presentation of executions by Moviola are the specific temporal relationships to be displayed and the time-scale to be used in the display. The user can define a subset of the interprocess relationships to be displayed. For example, the relationship between the process that writes a value and the process that subsequently reads the value is especially important. Similarly, in a message-passing model, the relationship between the process that sends a message and the process that receives it is important. In contrast, the relationship between a process that reads an object and the process that subsequently modifies the object is less relevant. Moviola allows the programmer to define classes of relationships implicit in the execution history to be highlighted in the display.

The time-scale for the display can be based on either logical time, local time, or global time. Logical time is a by-product of the partial order of an execution [Lamport,

1978]. The local time scale displays the height of an event in proportion to its duration according to the local clock of the process that recorded the event; the duration of events measured using different clocks are incomparable. A global time scale requires synchronized clocks. The clocks on the Butterfly run at essentially the same rate; our implementation exploits this fact to derive a global time scale.[4] Given a global time scale, the temporal relationship between events in all processes is determined, thereby defining their relative positions in the display.

Moviola's user interface provides a rich set of operations to control the graphical display. Several interactive mechanisms, including independent scaling in two dimensions, zoom, and smooth panning, allow the programmer to concentrate on interesting portions of the graph. Individual events can be selected for analysis using the mouse; the user has control over the amount and type of data displayed for selected events. The user can also control which processes are displayed and how they are displayed. By choosing to display dependencies for a subset of the shared objects, screen clutter can be reduced.

While the graphical interface of Moviola is a powerful tool for examining communication patterns, as well as providing easy access to performance information, it is cumbersome to gather detailed performance statistics about a program execution by using a mouse to select individual events in the execution graph for expansion. A simple program executing on a modest number of processors can generate such a large execution history that manual analysis, even with the assistance of interactive tools, can be daunting. Extensibility and programmability are provided by running all workstation tools under the aegis of Kyoto Common Lisp [Yuasa and Hagiya, 1985]. Tools can take the form of interpreted Lisp, compiled Lisp, or, like Moviola, foreign code loaded into the Lisp environment. Our programmable interface enables a user to write Lisp code to traverse the execution graph built by Moviola to gather detailed, application-specific performance statistics. The programmable interface is especially useful for performing well-defined, repetitive tasks, such as gathering the mean and standard deviation of the time it takes processes to execute parts of their computation, or how much waiting a process performs during each stage of a computation.

Multiprocessor Components

In our Butterfly implementation of the toolkit, all communication on the target multiprocessor goes through a centralized module, *Communications Central*, executing on the single Butterfly processor to which the network interface is connected. This module is responsible for managing all toolkit-related communication on the multiprocessor. In particular, Communications Central must gather execution information from the various processors that make up the multiprocessor, pass it on to the programmer's workstation and network file system, and forward programmer commands from the workstation to the appropriate processor.

[4] Our dependence on equal clock rates can be removed by reconstructing a global clock using algorithms found in [Duda *et al.*, 1987].

The *Monitor Module*, which exports a library of instrumented primitives which must be used by each application process to synchronize access to shared variables, is a toolkit component that is linked as part of each process in an application program. Since most concurrent programs use standard synchronization primitives, the cost of instrumentation is limited to the non-recurring effort of creating these libraries. The Monitor Module also implements data structures, shared with Communication Central, that allow the transparent, background migration of execution history data from local memory to mass storage.

7 Sample Analyses

This chapter presents sample analyses of several parallel programs that were performed using a prototype of the integrated toolkit described in chapter 6. The primary purpose of this chapter is to demonstrate the utility of the proposed fine-grain execution tracing model for both debugging and performance analysis. A secondary goal of this chapter is to demonstrate that the prototype toolkit facilitates exploratory analyses using these traces.

In this chapter, we examine executions of parallel programs for sorting and matrix manipulation. These applications were not developed to serve as production utilities, rather, they were developed as research prototypes to study large-scale parallel computation. To this end, they are limited in their input/output capability, but the core aspects of the programs (*i.e.*, the strategies for problem partitioning and communication) reflect reasonable approaches for utilizing large-scale parallelism.

The first section presents a case study of the development of a parallel sorting application using the toolkit. Through specific examples, we demonstrate that the information provided by our execution tracing model is useful for both debugging and performance analysis; furthermore, we how the graphical interface of the toolkit makes this information readily accessible. The second section presents a study of the execution behavior of two versions of a program that solves a system of linear equations using Gaussian elimination. In this section, we show that our dynamic execution traces enable a detailed understanding of execution performance. Analysis of this application demonstrates the utility of the programmable interface of the toolkit for gathering and analyzing performance statistics from execution traces.

7.1 Sorting

This section recounts specific experiences using our toolkit to develop a parallel program to implement odd-even merge sort on the Butterfly. We trace the development of the application, which implements a variant of Batcher's odd-even merge sort [Knuth, 1973, pp. 224–226], through three versions.

7.1.1 The Odd-Even Merge Sort Algorithm

The algorithm under study is a coarse-grain adaptation of an algorithm found in Ullman [Ullman, 1984, pp. 224–226] for odd-even merge sort on a butterfly network. This algorithm has a recursive structure. A single stage to merge 2 sorted lists, each of which is partitioned among 2^i processes, is shown in figure 7.1.

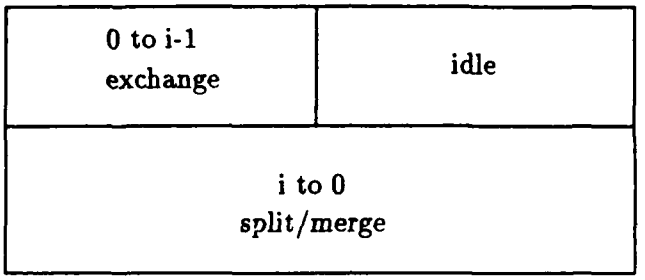

Figure 7.1: Structure of Merge Stage i.

The stage has two phases. In the first phase, processes with an *id* whose ith bit is zero participate in an *exchange*; other processes are idle for the first phase of that stage. In the second phase, groups of processes of size 2^{i+1} cooperate to form a sorted list spanning all processes in the group. The notation *0 to i - 1 exchange* refers to

$$\text{for } j := 0 \text{ to } i - 1 \text{ do exchange}(myid, \text{xor}(myid, 2^j));$$

where each process refers to its own id with the variable *myid* and exchange(a,b) denotes a swap between processes a and b of their sorted data. The sorting algorithm is asymmetric in that at merge stage i, only the processes with *id*s in which bit 2^i is 0 participate in the *exchange*; other processes are idle for the first phase of that stage. The notation *i to 0 split/merge* is shorthand for

$$\text{for } j := i \text{ to } 0 \text{ do split\_merge}(myid, \text{xor}(myid, 2^j));$$

In the *split/merge* phase of merge stage i, processes execute a round of *split/merge* operations with $i + 1$ communication partners. In a round of *split/merge*, each process sends the elements with even index in its own sorted list to its current communication partner (receiving in turn the elements of even index from the partner's list). Next each partner merges the remaining half of its sorted sequence with the sequence from its partner, splits the merged list into high and low halves and exchanges half of its list with its partner. The round completes with each partner merging the two list halves in its possession. The result of this round of *split/merge* is a sorted list that spans the pair of processes. In the base case of the recursive merge (merging 2 sorted lists, each contained by a single process), a single round of *split/merge* suffices, since the *exchange* phase degenerates to a no-op.

In our implementation, a master process spawns n slaves, where $n = 2^k$ for some k, assigning each slave a unique id in the range 0 to $n - 1$. The input data is partitioned among the slave processes; each slave manages its portion of the data independently.

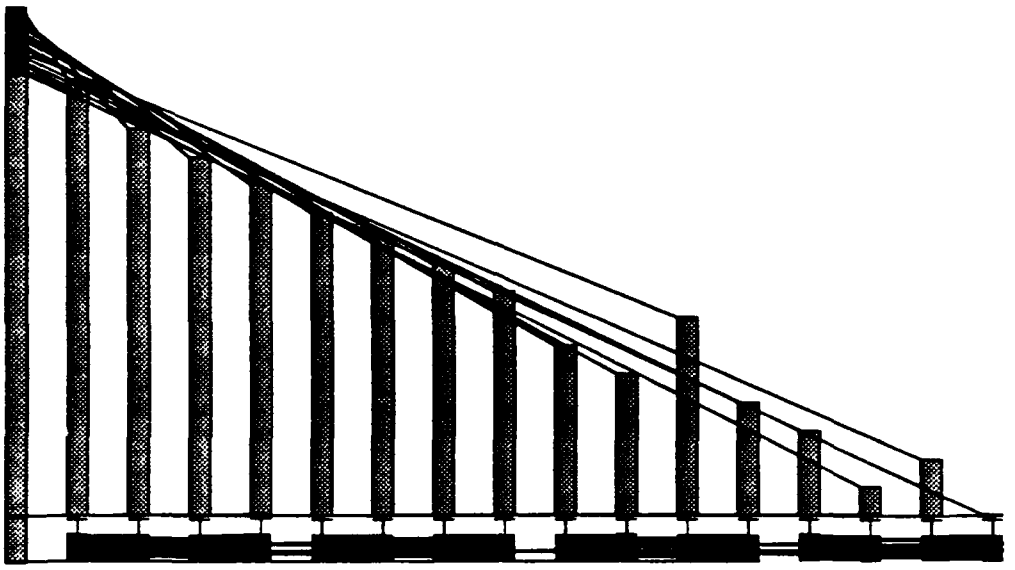

Figure 7.2: High-level View of a Parallel Sorting Program.

Each of the slaves sorts its own data locally, then the slaves cooperate to execute $\log_2 n$ merge stages to combine their portions of the input data into a single, sorted result. Communication among the slave processes is implemented using asynchronous message passing. Each process has a single buffer for incoming messages. *Send* waits until the target buffer is empty; *receive* waits until the target buffer is full.

7.1.2 Examining Execution Histories

Figure 7.2 is a high-level view of an execution of odd-even merge sort on 17 Butterfly processors.[1] The leftmost column in the figure shows a master process that runs on one processor; 16 slave processes on other processors are shown in the remaining columns of the figure. The diagonal lines emanating from the master to each slave represent a temporal relation induced by process creation. In this execution, each slave process obtains 500 integers from an input dataset and sorts them locally. The slaves then cooperate to merge the separate sorted lists into a single sorted list.

Our toolkit enables accurate measurement of the duration of each phase of the program execution. During startup, the master process sequentially creates its 16 child processes and the monitoring library opens a network connection for each process for recording execution trace data; this takes 9.15 seconds. It is necessary for the monitoring

[1] Recall from chapter 6 that processes are displayed in columns with time flowing downward. Each operation on a shared object by a process is noted by a hash mark in the column for that process. Arcs between processes denote temporal precedence of events. Shaded boxes indicate time a process was waiting.

library to dump execution trace data over the network since our Butterfly configuration has no local mass storage device. The cost of opening the network connections dominates the startup phase. The execution history graph in figure 7.2 dramatically illustrates Amdahl's law by showing how execution of a section of code with sequential constraints (network communication is centralized) dominates the execution time of the parallel program. In the second phase of the algorithm (the gap between the shaded boxes and the solid black band), each process sorts its portion of the data locally, which takes 289 *ms*. The merge phase of the algorithm (the black band of intense communication) takes 441 *ms*. Finally, the processes coordinate completion by signalling the master and moving their sorted data into a persistent result object.

For the remainder of this section, we focus on the merge phase of the sorting algorithm. This is by far the most interesting part of the executions under study, as it involves the most complex communication patterns and is highly parallel.

A magnification of the merge phase of the execution history is shown in figure 7.3. In this figure, *send/receive* pairs appear as diagonal edges from one process to another. Communication in the merge phase occurs in rounds. In each round, a process is paired with a partner. Each process sends to its partner, then receives the data sent by its partner. These communication pairs are visible in figure 7.3 as crisscrosses. In each merge stage of rank $i > 0$, the component *exchange* phase is visible in figure 7.3 as a butterfly network[2] of crisscrosses. This network is adjacent to 2^i shaded bars which correspond to processes in the right half of the merge stage waiting for processes in the left half to complete their exchange. Rounds of *split/merge* operations are visible in figure 7.3 as adjacent pairs of crisscrosses between two processes. The *split/merge* phase of a merge stage corresponds to a butterfly pattern that has two crisscrosses at each rank of the network.

7.1.3 Excerpts from a Debugging Session

Although a simple and elegant recursive formulation of the merging algorithm structures communication in a regular pattern (communication in the *exchange* and *split/merge* phases of each merge stage forms a pair of butterfly networks), it is not easy to use textual traces generated during program execution to check the correctness of communication patterns or to discover the cause of deadlock. Determining correctness or diagnosing an error using textual traces requires detailed hand simulation of the algorithm to determine which pairs of processes should communicate during each stage of the computation. Similarly, to discover the cause of an error from the state of the computation following a deadlock requires the same sort of hand simulation of the algorithm to determine how that state arose. However, the space-time diagrams of our toolkit illustrate the structure of communication patterns during a program execution and facilitate understanding program errors.

[2]Our use of the term *butterfly network* refers to a software communication structure among processes and should not be confused with the hardware implementation of that structure in the Butterfly multiprocessor.

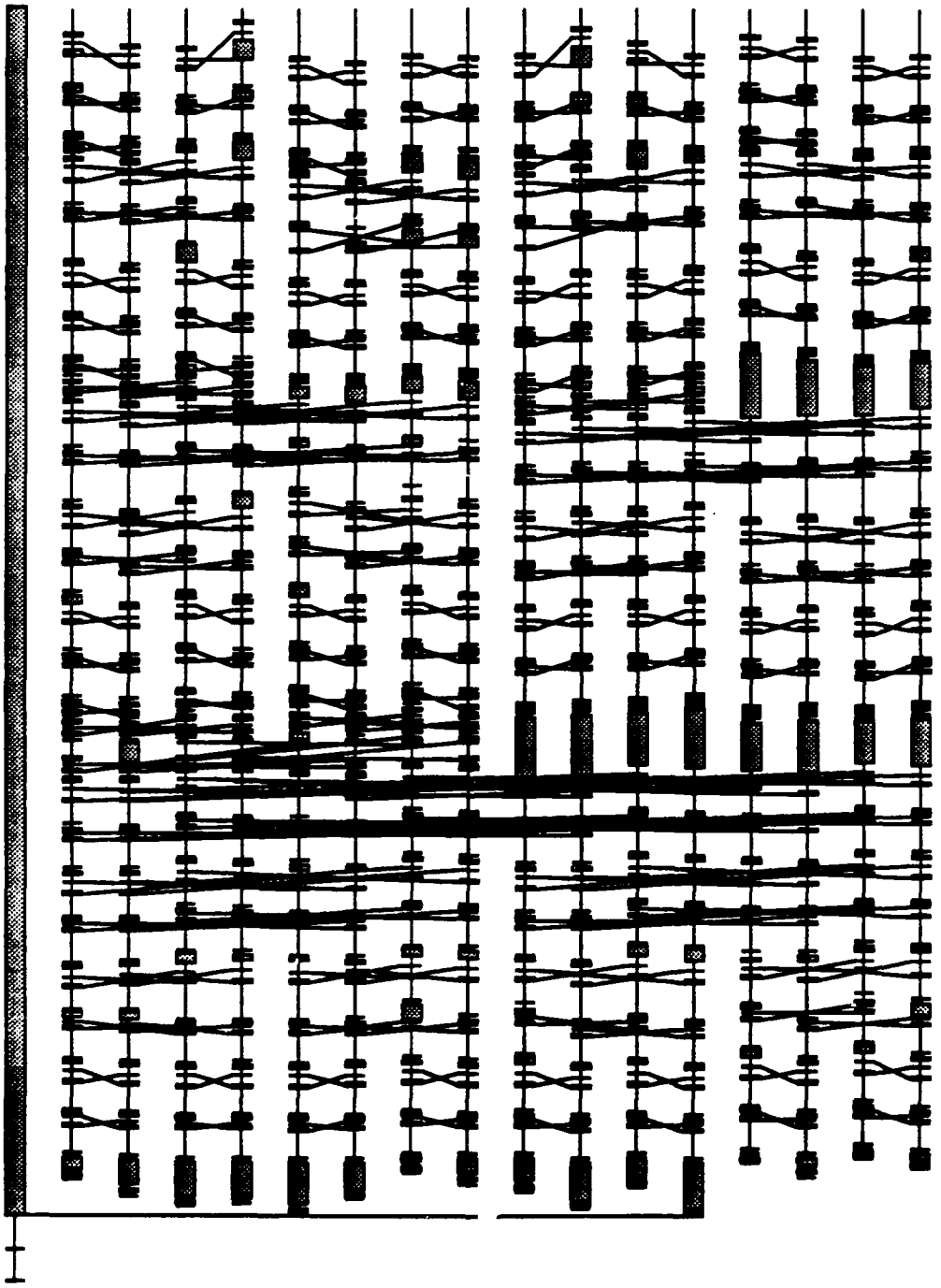

Figure 7.3: A Magnified View of the Merge Phase.

Figure 7.4 shows an execution history recorded early in the development of the odd-even merge sort program. This program incorrectly implements the recursive algorithm

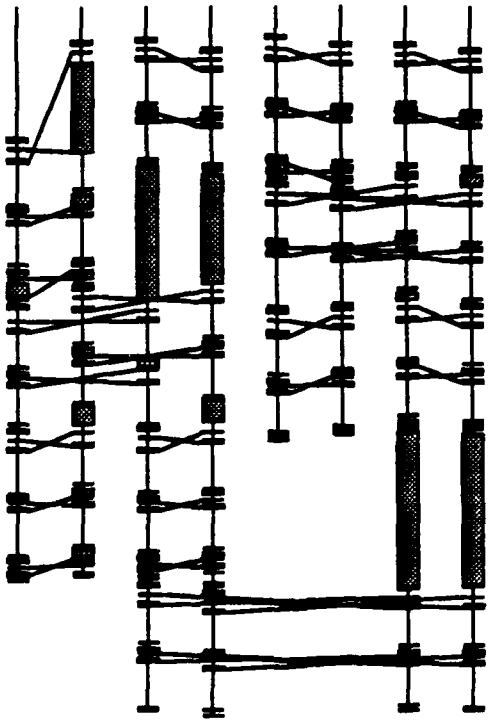

Figure 7.4: An Erroneous Merge Phase.

that specifies communication partners in each round of the merge stage; this causes executions of the program to deadlock. By comparing the execution history graph in figure 7.4 with the recursive formulation of the algorithm presented in section 7.1.1, it is clear that the program successfully completes merge stages 0 and 1. We can follow the progress of the execution by looking at slave processes 0 and 1, the two leftmost processes in the figure. The first two crisscrosses between these processes are the *split/merge* of merge stage 0. The next crisscross between process 0 and 1 is the *exchange* phase for merge stage 1. The next four crisscrosses for each of the processes 0–3 correspond to the *split/merge* phase of merge stage 1. During the *exchange* phase of merge stage 2, an error occurs. We expect to see a butterfly pattern (ascending from rank 0 to 1) between the 4 leftmost processes. We see the first stage of the pattern with crisscross pairs between (0,1) and (2,3) as the processes exchange with their partners at rank 0, then processes 2 and 3 erroneously communicate with processes 6 and 7 instead of completing the *exchange* phase at rank 1 with processes 0 and 1, respectively.

From this graphical view of the program execution, the error in the implementation of the recursive algorithm was easy to pinpoint. Clearly, the specification of which processes should be partners in an *exchange* phase was incorrect. Upon examining the implementation of the recursive merge algorithm in the program source code, it was

readily apparent that the error resulted from incorrect computation of a parameter to a recursive call of a routine invoked for a merge stage. The error appeared during merge stage 2, since earlier merge stages did not have a recursive call that uses this parameter (merge stage 0 had no *exchange* phase, and merge stage 1 had only a single round of exchanges).

Discovery and correction of this bug with our toolkit took under a half hour; in the past, other methods requiring hand simulation of the algorithm have required considerably more time and mental effort to discover the same type of error.

7.1.4 Performance Analysis

Our analysis toolkit provides easy access to information that enables detailed performance analysis. While using our programmable interface enables analysis of large-scale parallel programs with many processes, we expect that programmers will concentrate analysis efforts on executions with small numbers of processes (less than 32), as the value of performance analysis lies not in the size of the execution history under study, but rather in the thoroughness with which factors that affect scalability to different numbers of processors are studied. Close attention to scalability factors increases the predictive value of the performance model to executions manipulating larger amounts of data, or with different numbers of processes. In this section we present a sample performance analysis of the merge sort application accomplished with our tools, focusing on scalability predictions and performance tuning.

Using our toolkit to analyze an execution history with 4 slave processes, we measured the time required for a single *exchange* operation between two communication partners as ranging from 6.13 *ms* to 6.43 *ms* and the time for a single round of *split/merge* between two communication partners as ranging from 40.5 *ms* to 40.7 *ms*.[3] In terms of these quantities, the time to perform an n processor merge is predicted by the following recurrence:

$$\begin{aligned} T_1(n) &= k_1 \sum_{i=1}^{\log n} i + k_2 \sum_{i=1}^{\log n - 1} i \\ &= \frac{1}{2}\left[(k_1 + k_2)\log^2 n + (k_1 - k_2)\log n\right] \end{aligned}$$

where k_1 is the time for a single round of *split/merge* and k_2 is the time for a single *exchange*.[4] Using $k_1 = 40.6$ *ms* and $k_2 = 6.3$ *ms* (the mean times measured), we can predict the execution time for a scaled problem (each processor still responsible for 500 integers) on 16 slave processors: our equation predicts $T_1(16) = 444$ *ms*. Measuring the time for the merge phase of the execution shown in figure 7.2, we found the length of the merge phase to be 441 *ms* for a 16 processor execution; our prediction was accurate to within 1%. Detailed performance predictions of this kind are facilitated by the ability to easily and accurately measure the duration of parts of the computation with our toolkit.

[3] All times presented in this section are based on a clock with a resolution of 62.5 *us*.

[4] All logarithms are base 2.

The odd-even merge algorithm presented in section 7.1.1 is clearly unbalanced: in merge stage $i > 0$, the leftmost 2^i processes involved in each merge execute an *exchange* phase, while the rightmost 2^i processes wait until the *exchange* phase completes before they can begin the *split/merge* phase. In figure 7.3, the effects of the imbalance in the algorithm are visible as bars in the execution history of the processes on the right side of each recursive stage. These bars show graphically how long each process needs to wait for its communication partner at each stage of the computation.

We can determine the contribution of the *exchange* phase in each merge stage to the total execution time by examining the recurrence relation for the execution time. From the recurrence relation, the asymptotic contribution of the *exchange* phase accounts for $k_2/(k_1 + k_2)\%$ of the total execution time. For our measured values of k_1 and k_2, this is 13% of the execution time of the merge phase for problems in which each process manages 500 numbers. To accurately predict values for k_1 and k_2 for larger amounts of data, we merely need to examine two executions in which processes manage different amounts of data and measure the respective times for the *split/merge* and *exchange* operations. Using these data points we can write a linear equation that predicts values of k_1 and k_2 for all data sizes.

For the current data size of 500 elements per process, the 13% asymptotic overhead attributed to exchange operations merits attention during performance tuning. Since the *exchange* phase of the sorting algorithm results in an unbalanced computation, we can hope to achieve up to 13% better performance during the merge phase of our sorting program if we can shorten the *exchange* phase to provide more balance to the computation.

Examining the effect of the *exchange* phase on the data managed by processes, we note that the butterfly network of exchanges in merge stage i has the net effect of a data swap between each process with $id = b_{k-1}b_{k-2}\ldots b_{i+1}0b_{i-1}\ldots b_0$ and its partner with $id = b_{k-1}\ldots b_{i+1}0\overline{b}_{i-1}\overline{b}_{i-2}\ldots\overline{b}_0$.[5] We can make the *exchange* phase of each merge stage more efficient by replacing the butterfly network in the *exchange* phase by a point-to-point exchange which achieves the same result. The resulting execution graph of a program incorporating this improvement is shown in figure 7.5. Comparing figure 7.5 with figure 7.3, we see graphically how the improvement in the algorithm reduces the length of wait time for the rightmost half of processes in each merge stage; this in turn reduces the total execution time of the program. The modified recurrence relation that reflects the effect of the performance improvement is:

$$\begin{aligned} T_2(n) &= k_1 \sum_{i=1}^{\log n} i + k_2(\log n - 1) \\ &= \frac{k_1}{2}\log^2 n + (\frac{k_1}{2} + k_2)\log n - k_2 \end{aligned}$$

Asymptotically, the presence of the improved *exchange* phase now is vanishingly small. While this new program's performance scales satisfactorily, the algorithm can be restructured slightly to subsume the *exchange* phase in the *split/merge* phase. Figure 7.6

[5] For $n = 2^k$ slave processes, slave *ids* consist of k bits.

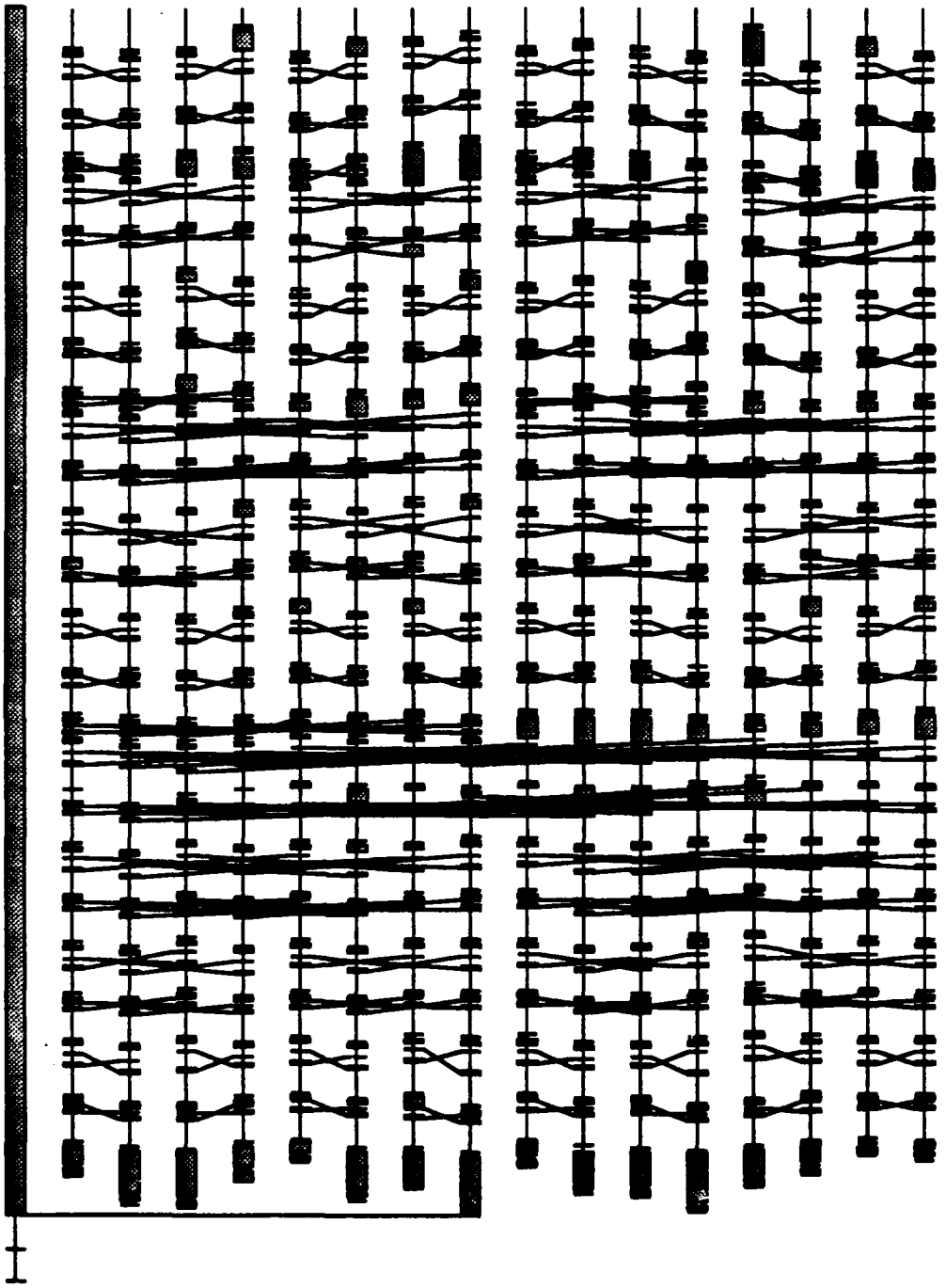

Figure 7.5: Merge Using a Single Round Exchange Phase.

shows an execution history in which the algorithm is balanced using this technique. Comparing this figure with figure 7.5, we see that the new algorithm again reduces waiting; in figure 7.6, processes wait in the merge stage only as a result of the inherent asynchrony of their executions. The performance of this final version of the sorting program obeys the recurrence equation:

$$T_3(n) = k_1 \sum_{i=1}^{\log n} i = \frac{k_1 \log n(\log n + 1)}{2}$$

Using the constants measured from the execution of the initial version of the program in figure 7.3, recurrence equation T_3 predicts the execution time of the merge phase of the improved program to be 406 *ms*. The measured execution time of the merge phase in figure 7.6 is 400 *ms*; our prediction is accurate within 1.5%.

7.2 Gaussian Elimination

Gaussian elimination is a well known technique for solving a system of linear equations (See Chapter 1 in [Strang, 1980] for background on this technique.) This section analyzes two parallel implementations of Gaussian elimination. Gaussian elimination consists of two distinct phases: upper triangulation of the input matrix, followed by back substitution. Since back substitution is largely a sequential activity, it is not particularly interesting as the object of a study of parallel program performance. For this reason, the Gaussian elimination programs under study here perform only the upper triangulation of the input matrix.

The implementations studied in this section do not use floating point arithmetic to manipulate the elements of the matrix since our largest Butterfly machine does not have hardware floating point support. The decision was made not to use software floating point as the computation time for row eliminations would dwarf the communication in the algorithm. Since the communication behavior of the program is an important factor in determining a program's scalability, integer addition and subtraction were used in place of the floating point multiply and divide so that the results would better reflect the ratio of communication to computation that would occur in machines equipped with floating point hardware.

The two programs under study here use a message-passing paradigm of communication. The original version of the program was written as part of a case study comparing shared-memory and message-passing communication paradigms [LeBlanc, 1986]. The following section describes the parallel algorithm shared by the two implementations. Section 7.2.2 describes a performance study of the original program accomplished using the toolkit. This analysis suggested modifications to the program to reduce communication cost and memory contention. A comparative analysis of the modified program is presented in section 7.2.3.

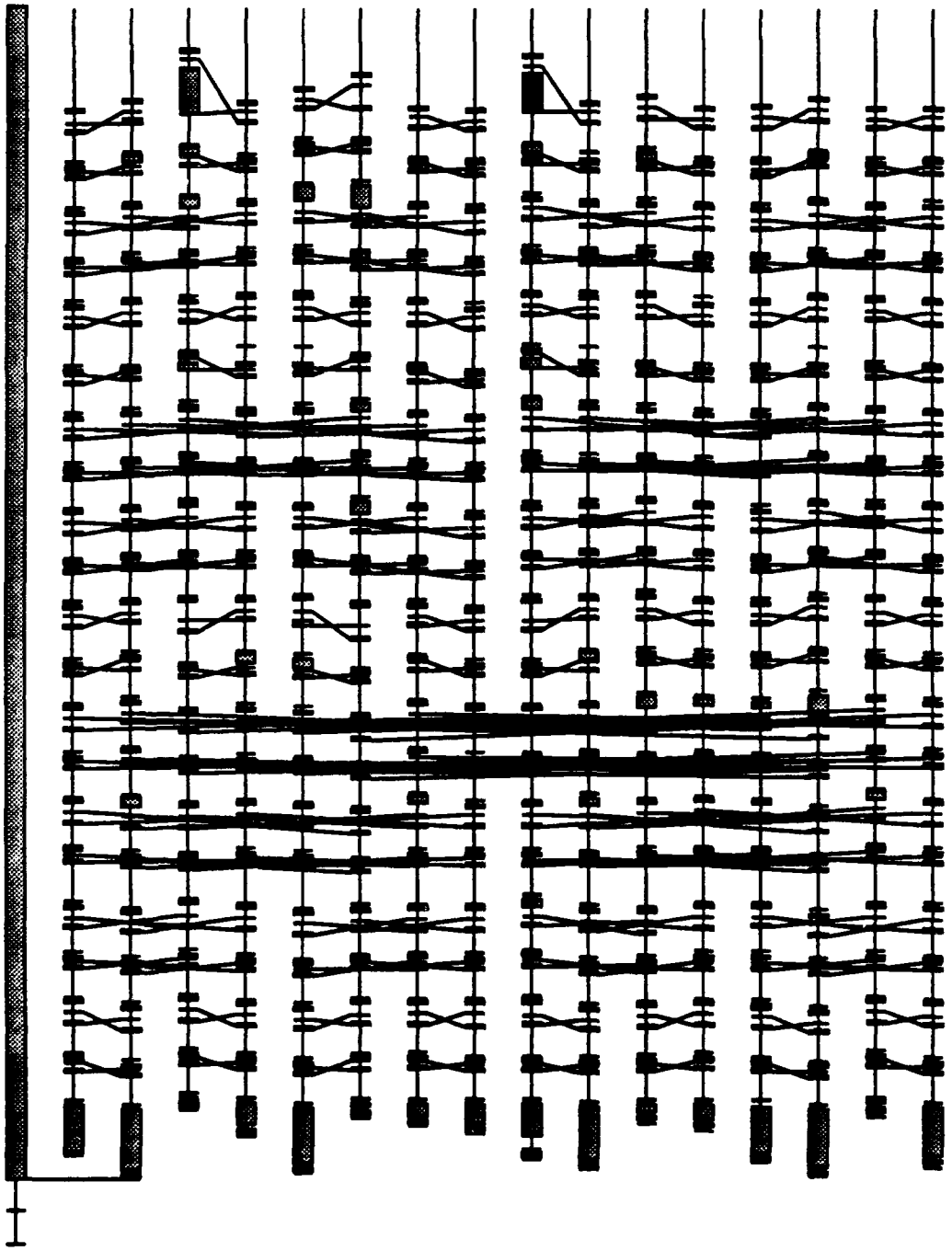

Figure 7.6: Elimination of the Exchange Phase.

7.2.1 The Upper Triangulation Algorithm

The basic step of the upper triangulation phase of Gaussian elimination involves subtracting a multiple of a row, which represents a linear equation, from each of the higher-numbered rows in the matrix to convert them into a simpler linear system with one less unknown. More precisely, for an $n \times n$ matrix, each row i of the matrix is used to eliminate the ith element from each of the rows $i < j < n$. This step has a natural parallel formulation, since each of these subtractions are independent.

```
numlocalrows := N/P
if (processor_id = 0)
    publish row 0 in a shared buffer
for column := 0 to N - 2
    obtain current pivot row from processor (column mod P)
    localrowstart :=  (column - processor_id + P)/P
    for row := localrowstart to numlocalrows
        fraction := pivot[column] / matrix[row,column]
        for i := column to N
            matrix[row,i] := matrix[row,i] - fraction * pivot[i]
        if (row = column - 1)
            publish the current row in a shared buffer
```

Figure 7.7: The Upper Triangulation Algorithm.

The algorithm used in the two programs of this case study partitions an $n \times n$ matrix among p processors by assigning rows in a modular fashion. Each processor i manages rows $0 \leq k < n$ for which $k \ mod \ p = i$. By assigning the rows in this manner, the work of using a pivot row to eliminate a column in all of the higher-numbered rows is partitioned evenly among the processors. Figure 7.7 shows the upper triangulation algorithm executed by each of the processes in the computation. Each row in the matrix represents a linear equation by the coefficients of the n variables and the constant term. Row manipulations are applied to both the coefficients and the constant term. In each step of the algorithm, each process gets the current pivot row and uses it to eliminate a column from all of the higher-numbered rows that the process manages. The first row needs no processing so it is immediately made available to each of the processes for use in an elimination step.

Communication between the processes in the computation uses a message-passing style that is efficient for distributed-memory multiprocessors. When a pivot row is complete, the process that manages it publishes it in an output buffer where it is available to other processors. When a process needs a row managed by another process, the row is obtained by copying it into a local buffer from the output buffer of the process that manages it. Copying each pivot row to a local buffer before using it in an elimination step is advantageous on the Butterfly since:

- a microcoded block copy primitive makes a single copy of a block of b elements

faster than remotely accessing each of the b elements individually,

- references to local memory are faster than references to remote memory, and
- accessing each element remotely only once reduces the potential for memory and switch contention.

The implementations of this algorithm studied in sections 7.2.2 and 7.2.3 are structured as a master process and a set of workers. The master process accepts as parameters p, the number of processors to be used, and n, the size of the matrix. Since the programs were intended as a vehicle for studying parallel computation rather than for production use, the programs synthesize their input data rather than reading it from a file. At startup, the master creates a single worker process on each available processor. Each worker performs its initialization code and synchronizes with the master. When all of the workers have completed their initialization code, the master releases them to begin the computation. When the computation is complete, workers resynchronize with the master and are subsequently destroyed.

7.2.2 Analysis of the Original Implementation

In this section, we analyze a 37 processor execution composed of one master process, and 36 worker processes as they perform an upper triangulation of a 900×900 matrix.

As the first step in analyzing the program we use the graphical browser of the toolkit to examine the gross characteristics of the execution.[6] Using the browser, we measure a startup phase of 13.43 seconds, followed by the computation phase which lasts the remaining 81.64 seconds of the execution. The startup phase includes the time to create all of the workers, have each worker synthesize its part of the problem matrix and resynchronize with the workers. As with the sort programs analyzed in section 7.1, this phase is dominated by the cost of opening a network connection for each process to dump execution trace data, so study of this phase will indicate little about the program's behavior. We focus our analysis on the computation phase in which the upper triangulation of the matrix is computed.

Figure 7.8 shows the first complete cycle in the computation. The leftmost process is the master, which is waiting for the workers to complete, and the remainder are worker processes. A complete cycle in the computation involves completing p columns of the upper triangulation (here, $p = 36$). The communication patterns in the figure clearly show the broadcast-like nature of the interprocess communication. In each round of the cycle, a process writes a pivot row in its output buffer, then, the other processes read that row and use it to eliminate a column in their unfinished rows in the upper triangulation. The edges between processes reflect reader/writer dependencies that are recorded when library routines for CREW access control of the communication buffers are used. The sloping edges from buffer *writes* to subsequent *reads* indicate that the workers are operating relatively asynchronously. If the computation were tightly synchronized, the dependency edges would be nearly horizontal.

[6] The entire execution is not shown.

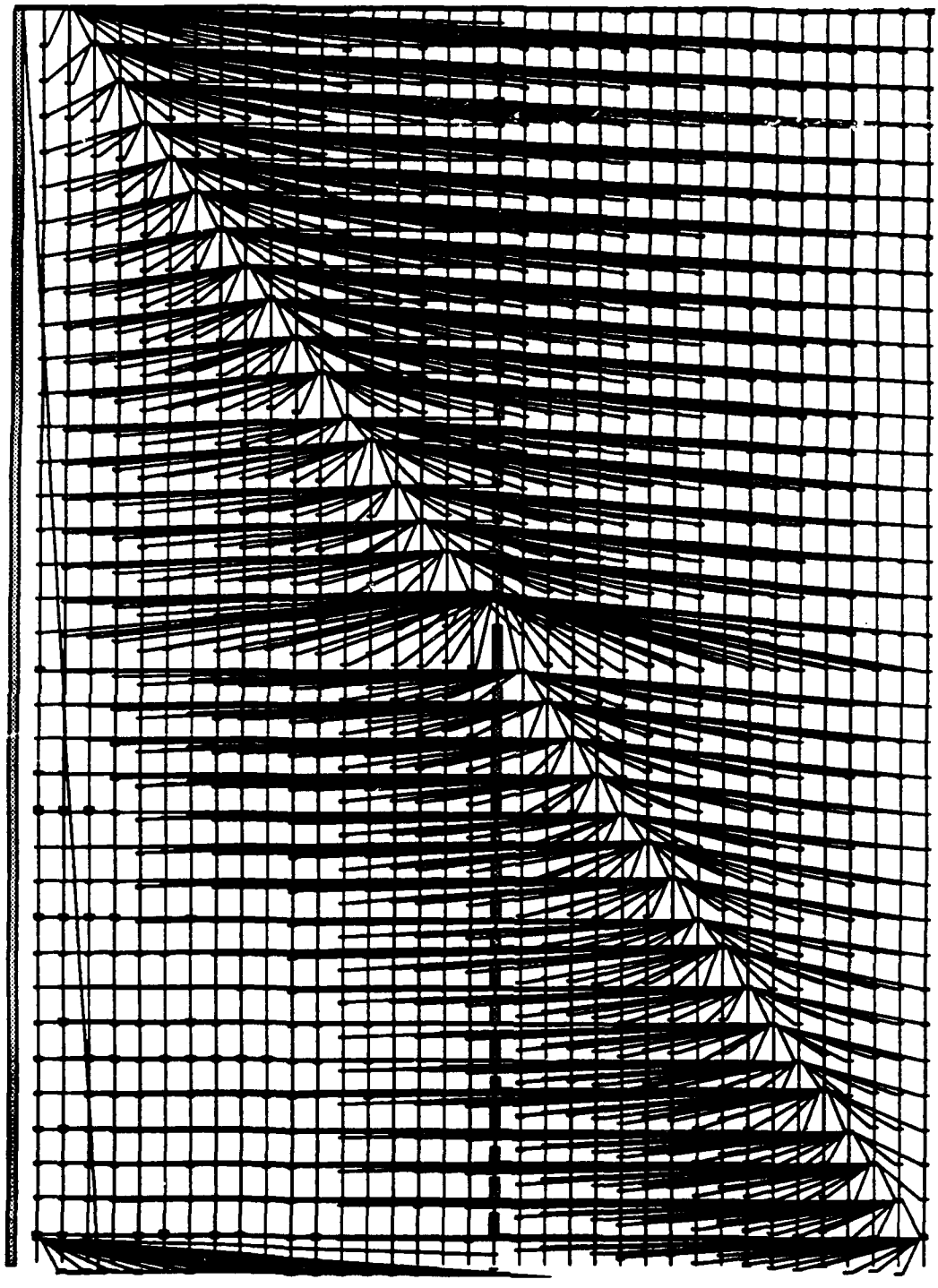

Figure 7.8: First Computation Cycle in Gaussian Elimination.

Figure 7.8 shows some anomalous behavior in the computation. The shaded bars in the history of the 19th worker process from the left show that the worker completes its elimination step of each round much faster than all of the other processes, and then waits for the next pivot row to become available. A visual inspection of the problem partitioning code in the algorithm showed nothing unusual. A check with the system administrator indicated that a faster processor board (a node with an MC68020 which has an on-chip instruction cache) had been added to the machine in that slot. Without a graphical representation of the computation, the pattern of the anomalous behavior would have been much less obvious.

As the rounds of the computation advance, the time for each process to complete its elimination step decreases, since processes consider only columns with index greater than the pivot row. At the same scale as figure 7.8, figure 7.9 shows the last 4 1/2 cycles of the computation. In comparison to the first cycle, here the processes are much more closely synchronized, as evidenced by the nearly horizontal dependency lines. Of some concern here, the figure shows shaded bars for all of the processes, indicating that processes are often spin-waiting for the next pivot row to become available. These shaded bars, although small, appear to account for a significant fraction of the time spent in these final stages of the computation and thus merit a closer examination.

An expanded view of a round in the final phase in figure 7.10 shows the behavior in question. The figure shows that all of the processes complete their row processing before the next pivot row is available and thus end up waiting, as indicated by the shaded bars. The event following the shaded bars in each of these processes marks the end of copying the pivot row to a local buffer. The interval between this event and the next set of shaded bars represents the time spent using the pivot row to eliminate columns in the remaining rows. This figure clearly shows that, in the final rounds, communication time dominates the time spent in the elimination step. Also, this figure shows that the computation is now quite synchronous and many processes need the pivot row at the same time; this leads us to suspect that memory contention may be present in the late cycles of the computation.

While the graphical views of the execution furnished by the interactive browser provide us with a general understanding of the behavior of the Gaussian elimination program, in isolation they are insufficient to gauge the need for (and the potential impact of) performance tuning. For more detailed analysis, we use the programmable user interface to write functions in Common Lisp to gather and analyze data captured in the execution trace. (Appendix C contains all of the Lisp code used to collect data from the execution histories to generate the graphs in this section and the following section.) This data can either be examined directly in textual form, or exported from the toolkit in formats suitable for input to other analysis packages.

The scalability of a parallel program, often measured in terms of *speedup*, ultimately depends on what fraction of execution time a parallel program spends directly working on the problem to be solved compared to the fraction it spends on parallelization overhead. One form of parallelization overhead that has a significant impact on speedup is overhead due to communication. The ratio of communication to useful computation in a parallel program execution puts an upper bound on the speedup achievable. To de-

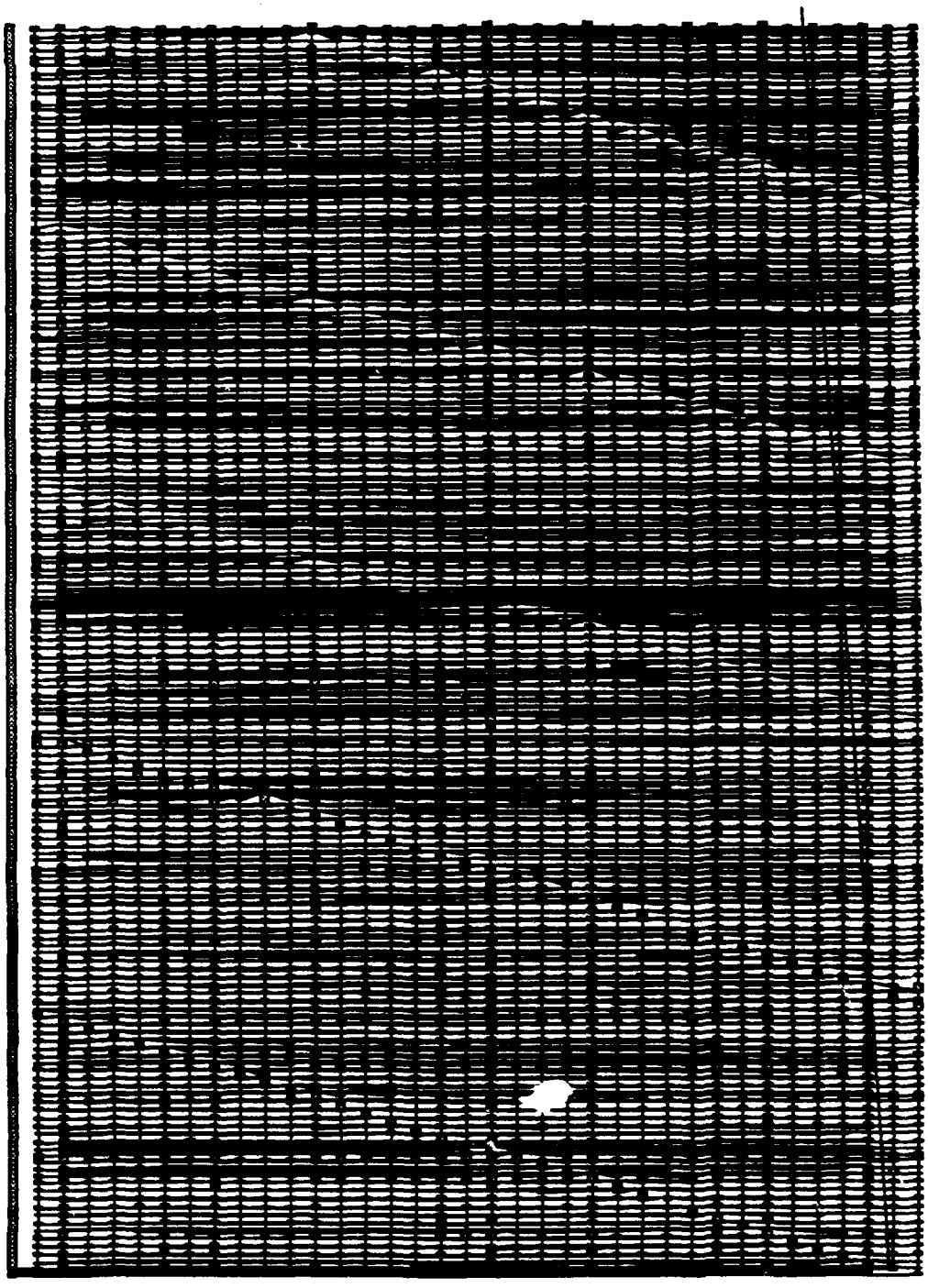

Figure 7.9: Final Computation Cycles in Gaussian Elimination.

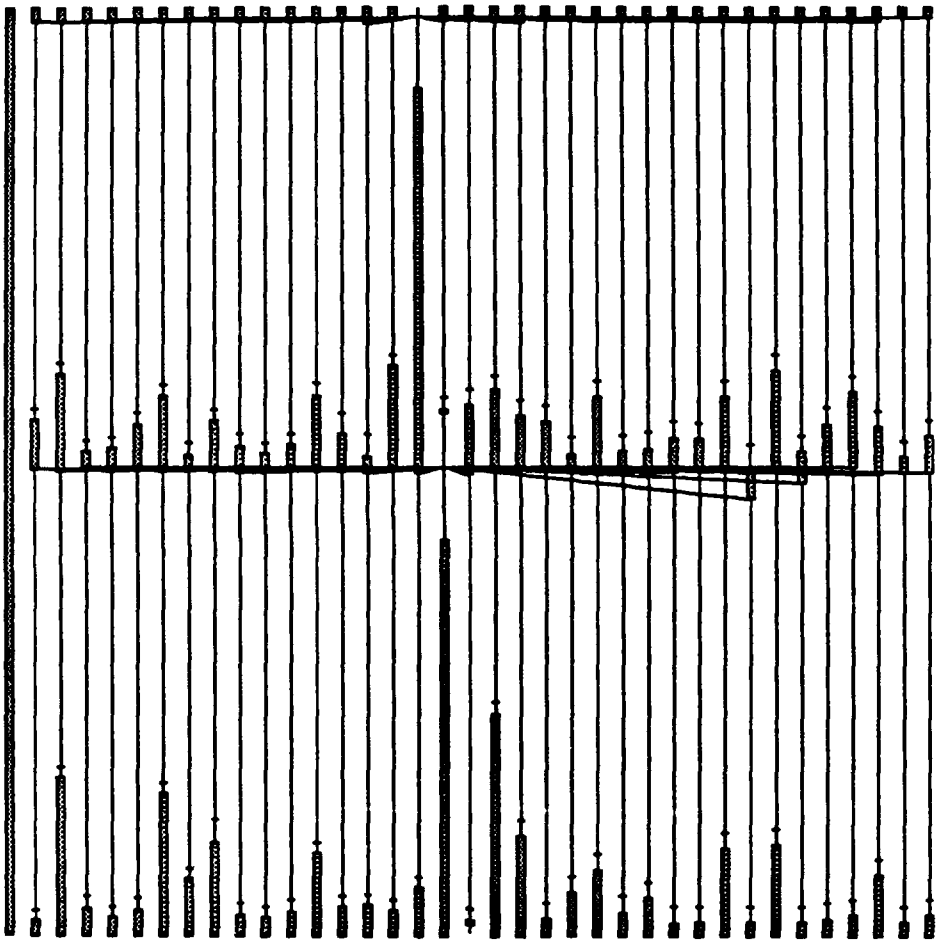

Figure 7.10: A Single Round in the Final Cycle of the Upper Triangulation.

termine the fraction of the execution that the Gaussian elimination application spends communicating with respect to the total time spent computing the upper triangulation, we wrote Lisp routines that sift through the program execution traces summing the intervals of communication during the computation phase and dividing the resulting value by the total length of the computation phase. The information collected by these routines was used as input to S [Becker and Chambers, 1984], an interactive environment for data analysis and graphics.

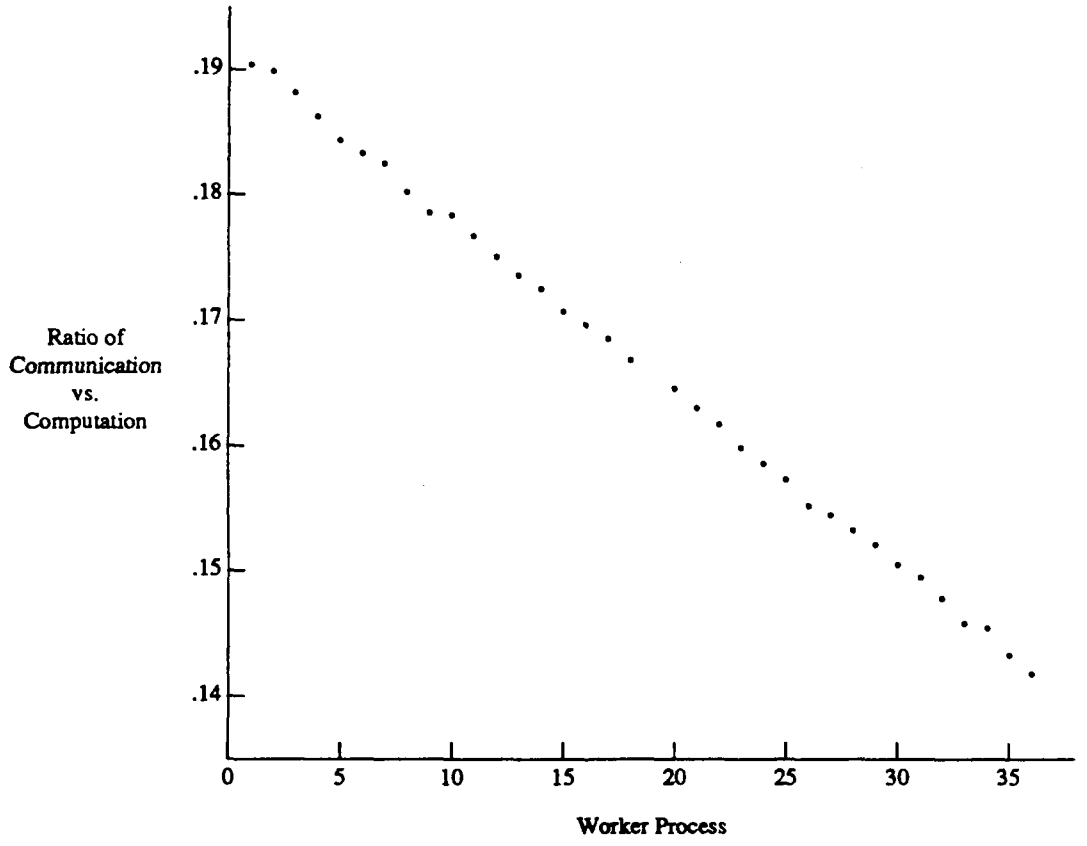

Figure 7.11: Ratio of Communication vs. Computation in Parallel Gaussian Elimination.

Figure 7.11 shows a plot of the ratio of communication to computation in the upper triangulation phase for each worker process in the execution. We omit data for worker process 19 in this and all subsequent figures since including its outlying data requires expansion of the graph axes which obscures the significance of the relationships present in the other data.[7] The information in this graph tells us two things. First, a substantial fraction of the execution time of the computation is being spent waiting for pivot rows to become available and then copying them between processors. From the structure of the algorithm, we know that the fraction of the execution spent on communication will increase if the algorithm is used to solve a problem of the same size on a larger number

[7] Recall from above that processor 19 is faster, which accounts for its outlying data points.

of processors; this limits the speedup achievable.[8] Second, the graph shows a puzzling linear trend in the communication ratio that ranges from a high of 19% for the first worker to a low of 14% for the last worker. From the structure of the algorithm, in which each processor is responsible for n/p rows, the initial expectation was that the ratio of communication to computation would be identical for each of the processes.

To examine the differences in the worker process's dynamic behavior that result in the decrease in communication ratio with increasing processor number, we wrote Lisp code for the toolkit's programmable interface that gathers a complete trace of the communication cost incurred by each processor in each round. The impact of the trend towards less communication with increasing virtual processor number can be best observed by examining communication traces for a processor at each extreme. Figure 7.12 is a plot of the time that worker 1 spent publishing or copying a pivot row in each round of the computation. Figure 7.13 plots the same information for worker 36. Each communication time displayed in these plots corresponds to the length of the interval between the time the worker requested a lock guarding access to the output buffer containing the current pivot row, and the time the lock was released. Thus, the communication time includes both spin-wait time prior to gaining access to the pivot row in the shared buffer, and the actual time to transfer the pivot row data between lock acquisition and release.

The difference between the two workers in their dynamic communication behavior is striking. In early rounds of the computation, worker 1 spends as long as 88.75 *ms* completing the communication for a round. In comparison, worker 36 never spends over 13.25 *ms* completing any of the early rounds of communication. Near the end of the computation, however, the range of communication times for both workers appears comparable. While these plots clearly differentiate the dynamic behavior of the two workers, to understand the cause of the imbalance of communication, it is necessary to look at another level of detail to find out whether the communication imbalance results from spin-waiting or memory contention.

A Lisp routine nearly identical to that used to collect the communication trace was used to collect the trace of the data transfer time in each communication round. (The data transfer time corresponds to the duration that the lock was held on the buffer containing the pivot row during each round.) Figure 7.14 shows that in the early rounds of the computation, the data transfer time accounts for about half of the difference between the communication time of worker 1 and worker 36. In the later rounds, the data transfer time appears to account for nearly all of the communication time for worker 1. Since each worker executes identical code and the amount of data transferred in each round is constant for the entire computation, the increase in data transfer time indicates memory or switch contention.

In the later rounds, contention is severe for all workers. Inspection of the algorithm shows the probable cause for this behavior. As execution progresses, the amount of

[8] Gustafson [Gustafson, 1988] argues that scaling the number of processors without proportionally scaling the size of computation gives an unfair measure of speedup. However, even with a proportional scaling of the problem, the ratio of communication to computation increases on a larger number of processors for this algorithm.

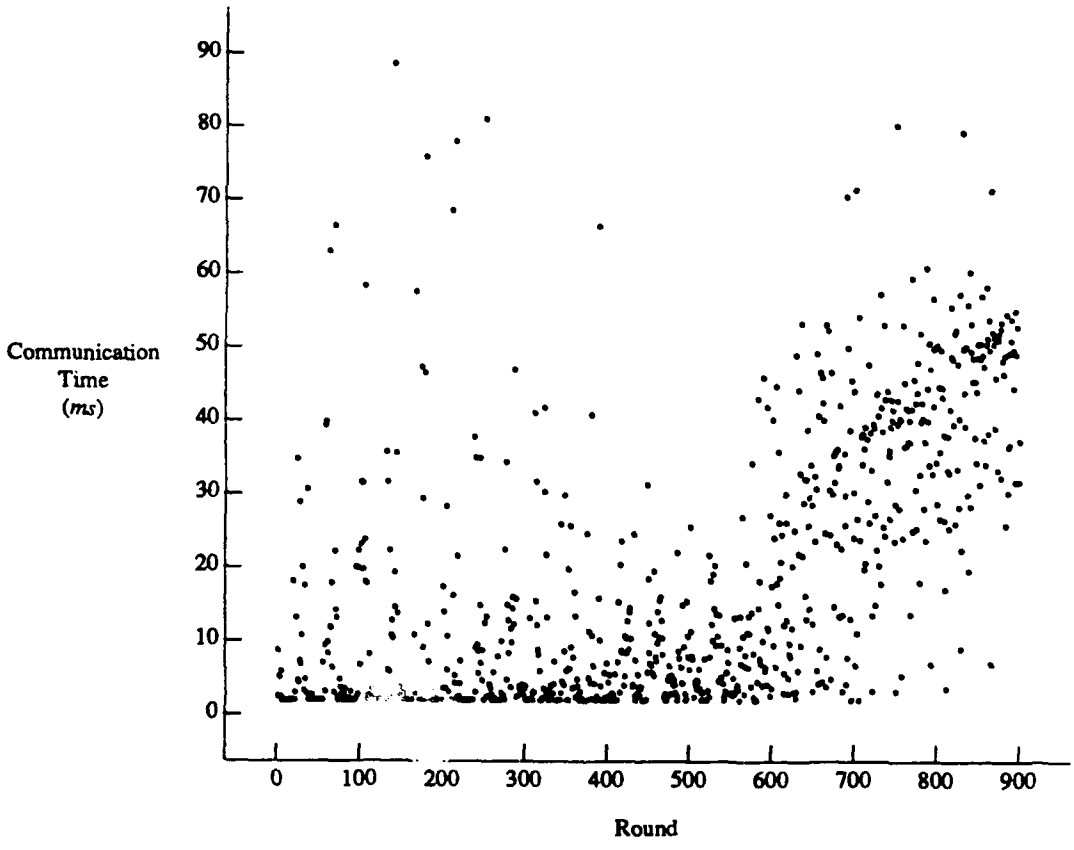

Figure 7.12: Communication Time per Round for Worker Process 1.

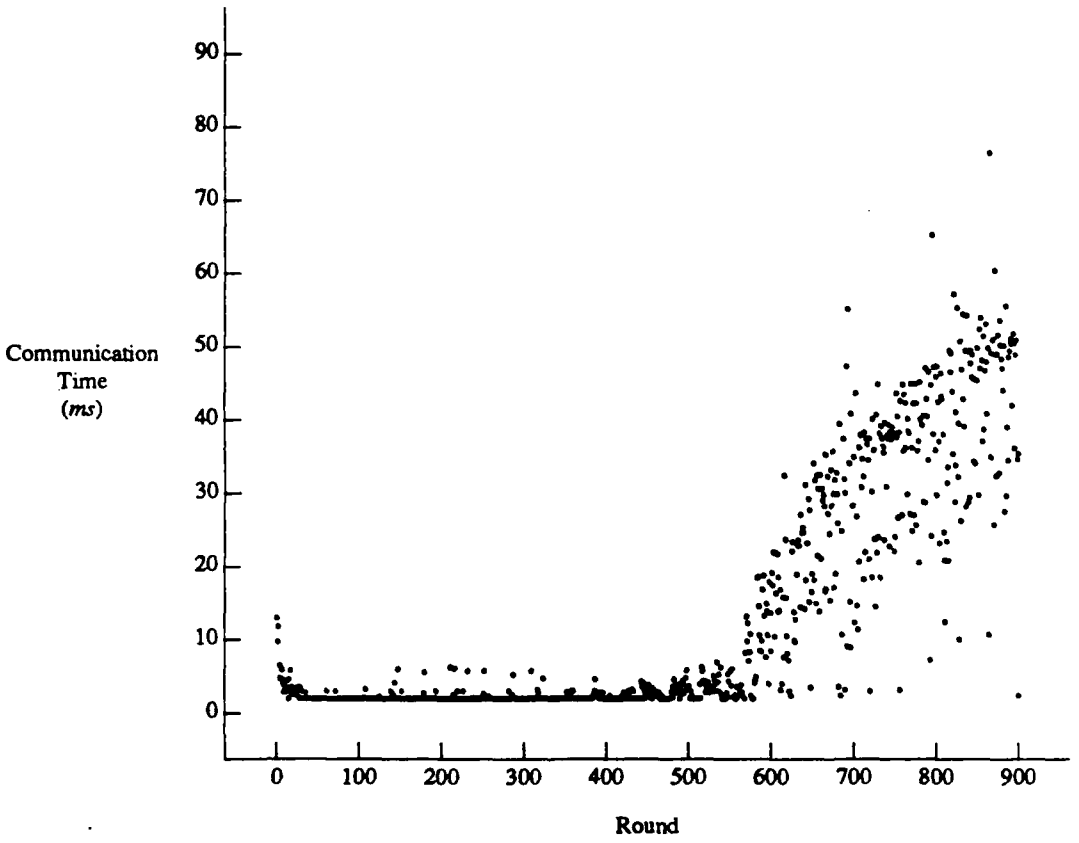

Figure 7.13: Communication Time per Round for Worker Process 36.

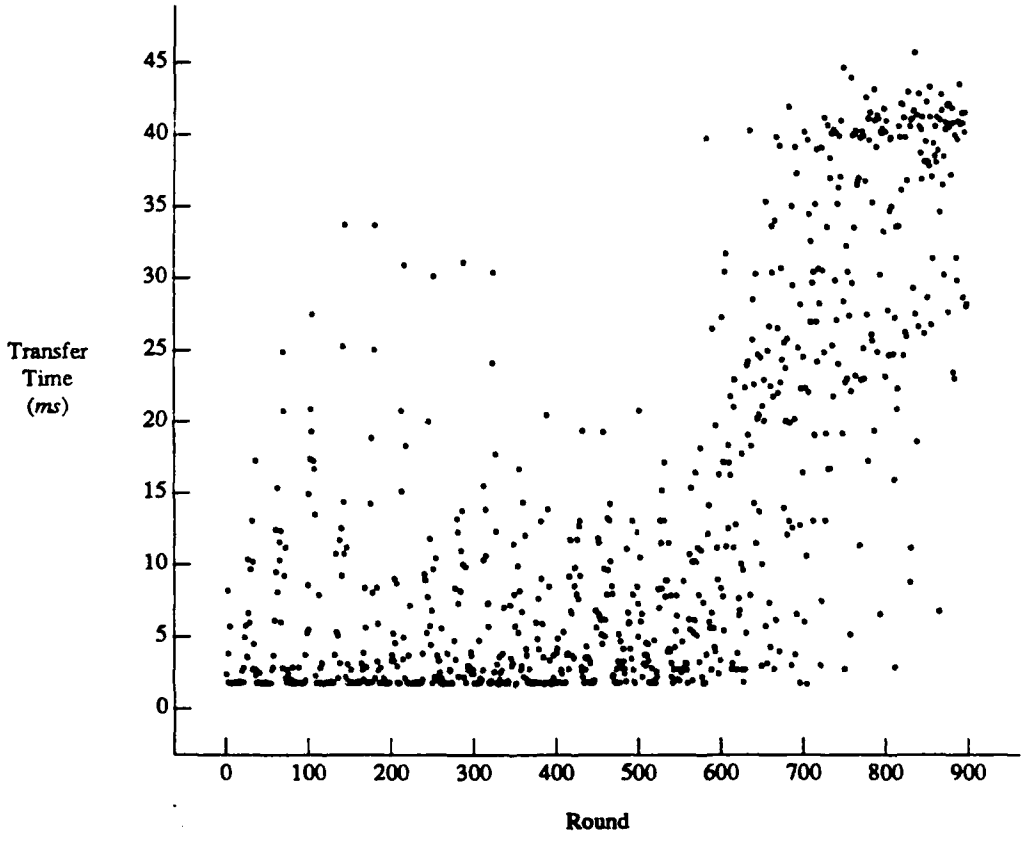

Figure 7.14: Pivot Row Data Transfer Time per Round for Worker Process 1.

computation in each round decreases (since the size of the linear system shrinks by one each round); however, the amount of communication per round (copying the pivot row to a local buffer) is fixed. Thus, as the execution progresses, the ratio of communication to computation per round increases dramatically; in later rounds, the communication time dominates the computation time. This explains the behavior shown in figure 7.10: while the processor with the pivot row publishes it in his output buffer, all other processors spin-wait until the row is available. Once the row becomes available, all of the waiting processors attempt to transfer the data simultaneously, resulting in severe memory contention. The minimum pivot row transfer time shown in figure 7.14 is 1.8 *ms*. In the final rounds the transfer times are clustered around 40 *ms*. If all 35 of the waiting processors simultaneously attempt a transfer, on average we would expect that any particular processor would need to wait for half of the transfers to finish. Thus, we predict the expected data transfer time in the final rounds to be $\frac{(p-1)}{2} \times (min\ transfer\ time) = \frac{35}{2} \times 1.8ms = 31.5ms$. These predicted transfer times are 30% lower than the times observed, however, they do account for a large part of the observed twenty-fold increase in transfer time.

Examination of the upper triangulation algorithm and its implementation indicated that we can reduce the amount of communication per round. In the implementation of the upper-triangulation algorithm, its designers overlooked the fact that the algorithm uses only the pivot row elements to the right of the diagonal. Instead of having each processor copy the entire pivot row into a local buffer during each round, processors need only copy the elements to the right of the diagonal. Fortunately, examination of the execution traces of the program made it apparent that a performance problem was present and helped us pinpoint its cause to a correctable error.

7.2.3 Analysis of the Improved Implementation

In this section, we analyze an improved implementation of the upper triangulation algorithm which reduces the number of pivot row data elements transferred by copying only the elements in the pivot row to the right of the diagonal. This modification is important since it improves the scalability of the program to larger problem sizes and larger numbers of processors. Without this improvement, increasing the problem size and/or the number of processors would exacerbate the existing contention.

In absolute terms, a 37 processor execution of the improved program completed an upper triangulation of the same 900 x 900 input matrix in 71.35 seconds, 12.6% faster than the original implementation. The goal of our program improvement was to reduce the memory contention encountered in the last half of the computation. Figure 7.15 shows a trace of the communication time per round for processor 1 in an execution of the improved program. In comparison to figure 7.14, we see that the data transfer time in the later rounds has been dramatically reduced by correcting the imbalance of communication to computation in later rounds by reducing the size of pivot row transfers.

Although our improvement reduced the high cost of communication in the later rounds of the computation, figure 7.15 shows that the disparity in communication costs

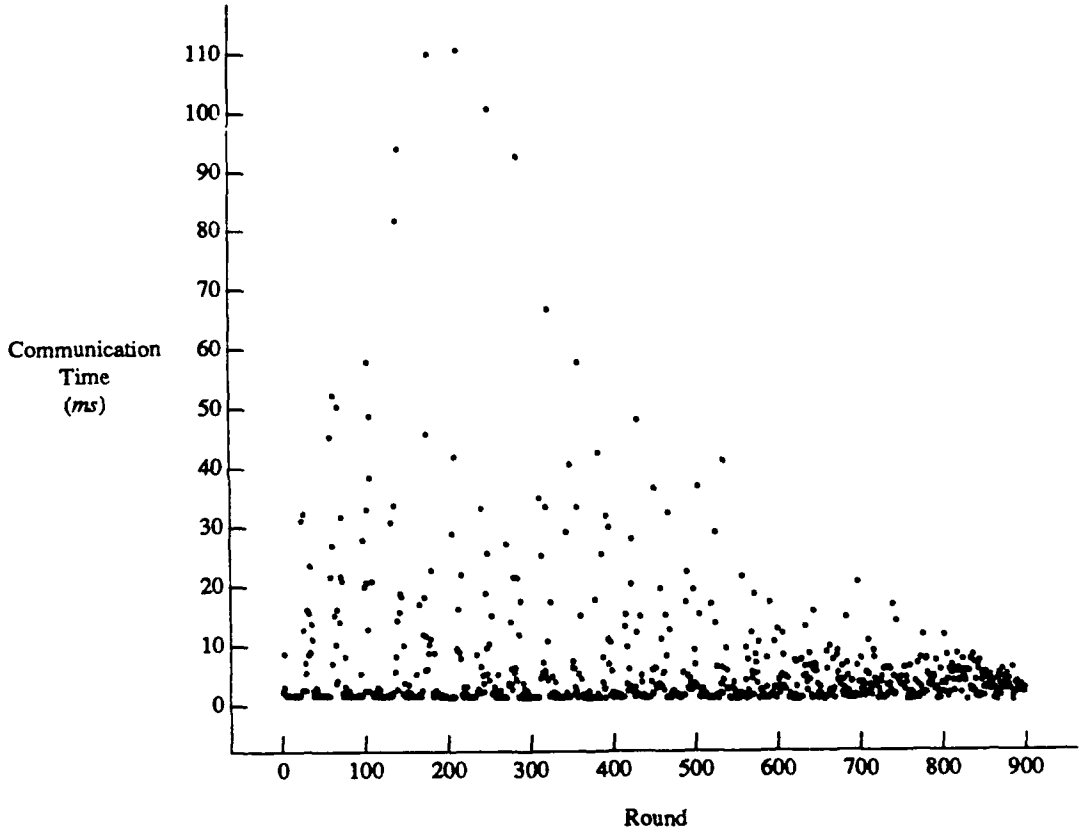

Figure 7.15: Communication Time per Round for Worker Process 1 (Improved Program).

in the early rounds, which we first observed for the original program in figure 7.12, is still present. At the scale of figure 7.15, it is not apparent if there is any structure inherent in this behavior. Figure 7.16 shows the data transfer time in the first 150 rounds of an execution of the improved program. From this figure, it is apparent that contention occurs in the early rounds at regular intervals with a period of 36, the number of worker processes.

Based on our static partitioning of the problem, our expectation was that the computation would be balanced with each processor operating on the same number of rows. No contention was expected in the early rounds of the computation. Comparing the communication times shown in figure 7.15 with the data transfer times shown in figure 7.16, we see that the data transfer time accounts for only about half of the communication cost; the difference is time spent spin-waiting. The presence of significant spin-waiting in these early rounds indicates that the the computation is not as balanced as we had initially thought. Apparently, pivot rows are not always ready when workers need them. Near the end of each 36 round cycle, worker 1 apparently "catches up" to the worker computing the pivot row and must wait until the pivot row computation is completed. For contention to occur, other processors must be "catching up" to the processor computing the pivot row as well. When the pivot row finally becomes available, all waiting processors initiate a pivot row transfer at the same time.

Figure 7.17 shows a perspective plot of the transfer time per round for all of the worker processes for the first 150 rounds of the computation. Rounds in the computation flow from foreground to the rear of the plot (along the Y axis). Pivot row data transfer time in each round is plotted in the Z direction. Workers are distributed along the X axis with worker 1 at the left and worker 36 at the right. From this figure, we can see how contention periodically builds during each cycle of the computation. A triangular wavefront of contention rises above the plane that corresponds to minimum transfer time. As the rounds progress, the number of workers that experience contention (those that catch up to the processor computing the pivot row) increases.

This view of the computation caused us to reconsider the assumption that the work is evenly balanced among the workers. Upon close examination of the algorithm, the cause for the imbalance becomes apparent. When a worker finishes a pivot row, it now has one fewer row to perform eliminations on in each subsequent round of the computation. Each higher numbered worker does not reduce the number of rows it performs elimination steps on until it produces its next pivot row. Thus, workers of lower index than the worker computing the current pivot row have less work in the current round than workers with higher index. A second order effect that reduces the impact of the row imbalance is the monotonic decrease in the length of the rows manipulated as the computation progresses. As the length of rows decreases, the work needed to maintain an extra row also decreases. Below, we describe equations that govern the imbalance in computation.

For the upper triangulation of an $n \times n$ matrix by p processors using the algorithm given in figure 7.7, we can calculate the amount of the imbalance in the computation that occurs during the cycle ending in round r (the last $r - max(r-p, 0)$ rounds). This imbalance $I(i,n,r,p) = max(E_P(n,r,p) - E_N(i,n,r,p), 0)$ is the positive difference of

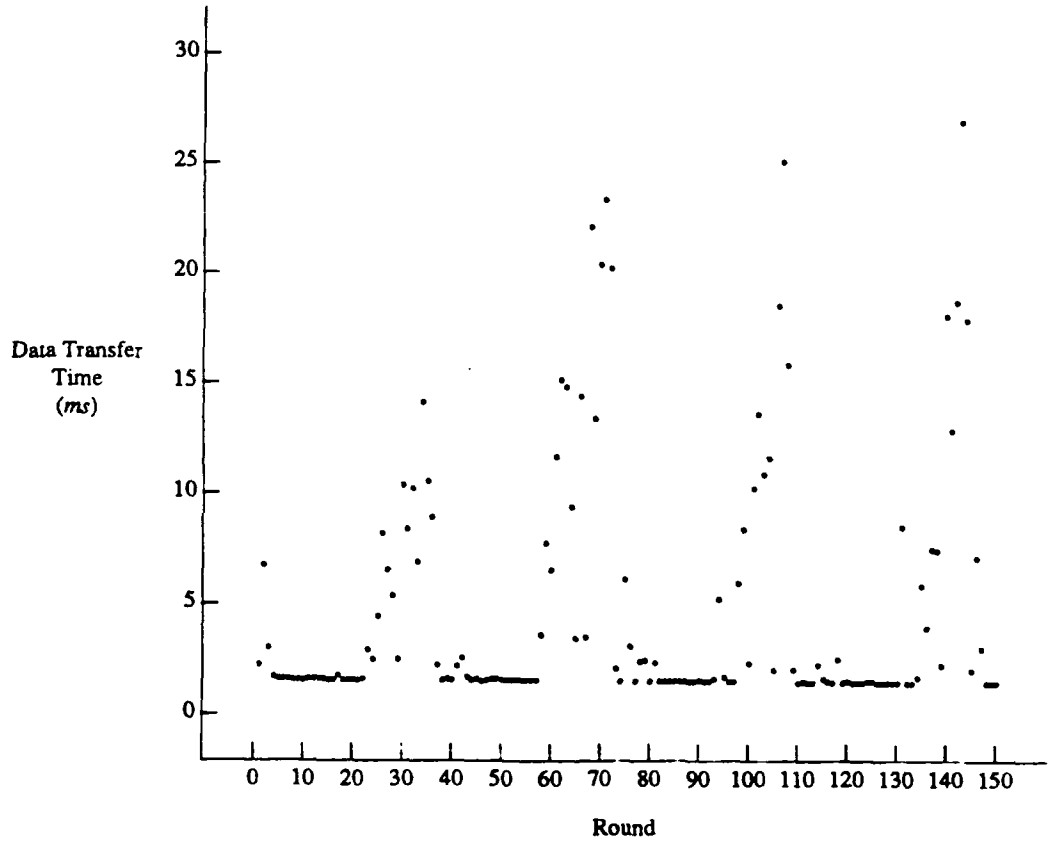

Figure 7.16: Pivot Row Data Transfer Time per Round for Worker Process 1 (Early Rounds).

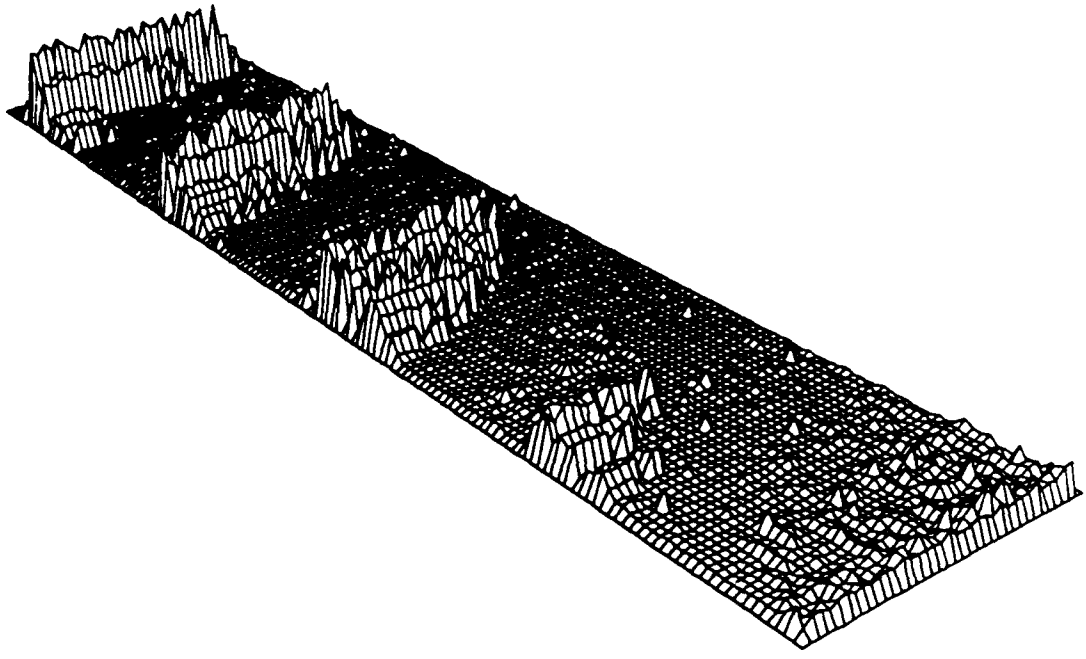

Figure 7.17: Pivot Row Data Transfer Time per Round for All Workers (150 Rounds).

two quantities: $E_P(n,r,p)$, the number of elements that processor ($r \bmod p$) has to perform elimination steps on between the time it publishes pivot row $max(r-p,0)$ and the time it publishes pivot row r, and $E_N(i,n,r,p)$, the number of elements processor i has to perform elimination steps on from the time pivot row $max(r-p,0)$ is obtained from processor $\frac{r}{p}$ (the last time processor i and processor $\frac{r}{p}$ synchronized since the beginning of the computation) until pivot row r is needed from processor $\frac{r}{p}$.

$E_P(n,r,p)$ is described by the following equations:

$$E_P(n,r,p) = \begin{cases} n-r+1+\left(\frac{n-(r \bmod p)+p-1}{p} - \frac{r}{p}\right)\sum_{j=r-p}^{r-2}(n-j) & r \geq p \\ n-r+1+\frac{n-r+p-1}{p}\sum_{j=0}^{r-2}(n-j) & 1 < r < p \\ n & r = 1 \\ 0 & r = 0 \end{cases}$$

Since pivot row 0 can be published unchanged, $E_P(n,0,p) = 0$. Pivot row 1 can be published as soon as the first column is eliminated using pivot row 0, which requires an elimination step for each of the n elements in the row 1. For pivot rows r, $1 < r \leq p$, an elimination step is required for each column to the right of the diagonal in each row managed by process r. The first term, $n-r+1$, is the length of pivot row r upon which a row elimination must be performed using pivot row $r-1$ before pivot row r can be published. The second term is the product of the number of rows that process r is managing and the number of columns remaining in each of the rows for rounds j, $0 \leq j \leq r-2$. Similarly, for pivot row $r \geq p$, the number of elimination steps performed between the time row $r-p$ was published and the time row r can published is the sum of two terms. As before, the first term, $n-r+1$, is the length of pivot row r upon which a row elimination must be performed using pivot row $r-1$ before pivot row r can be published. The second term is the product of the number of rows managed by process r that require row eliminations in rounds $r-P$ through $r-2$ and the number of columns remaining in each of the rows for rounds j, $r-p \leq j \leq r-2$.

$E_N(i,n,r,p)$ is described by the following equations:

$$E_N(i,n,r,p) = \begin{cases} \sum_{j=r-p}^{r-1}\left(\frac{n-i+p-1}{p} - \frac{j+p-i}{p}\right)(n-j) & r \geq p \\ \sum_{j=0}^{r-1}\left(\frac{n-i+p-1}{p} - \frac{j+p-i}{p}\right)(n-j) & 0 < r < p \\ 0 & r = 0 \end{cases}$$

In round 0, processor i needs pivot row 0 immediately before performing any elimination steps. Row r, $0 < r < p$ is needed after elimination steps have been performed on each of the rows managed by processor i for pivot rows 0 through $r-1$. The first term in the summation is the number of rows managed by processor i in each round j, $0 < j < (r-1)$. The second term is the number of columns remaining in each of the rows for round j, $(r-p) \leq j \leq (r-1)$. For round r, $r \geq p$, the equation describing the number of elimination steps to be performed before processor i needs pivot row r is identical to the equation for rows $0 < r < p$, with the exception of the lower limit of the summation which marks the beginning of the cycle.

Figure 7.18 plots $I(i,n,r,p)$, the imbalance in the computation between each of the processors and the processor computing the current pivot row, for the first 150 rounds

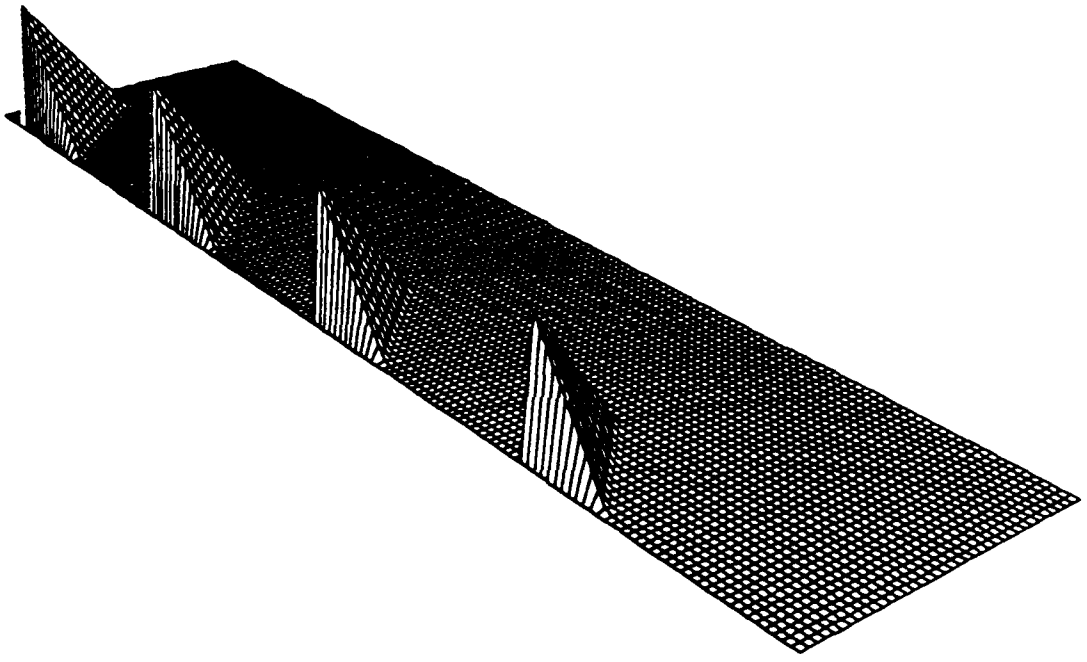

Figure 7.18: Imbalance in Computation between each Worker and the Worker Producing the Current Pivot Row (150 Rounds).

of the computation. Comparing the imbalance shown in figure 7.18 with the plot of the data transfer time shown in figure 7.17, we see a strong resemblance. The peaks in figure 7.18, which represent imbalance in the computation, occur with the same period and have the same general form as the peaks in figure 7.17, which represent contention during data transfer. Apparently, the imbalance between workers is a primary cause of memory contention in the early rounds of the upper triangulation. The imbalance in the computation in a round causes workers to delay until the current pivot row becomes available; when it becomes available, many workers simultaneously transfer the pivot row causing contention. Differences between figure 7.18 and figure 7.17 are likely due to two factors which are not accounted for in the analytic model: waiting in earlier rounds in a cycle reduces the imbalance in later rounds in that cycle, and the cost of transferring pivot row data (which grows proportionally larger in the presence of contention). The first factor should account for the flatter character of the peaks in the empirical data. The second factor should account for the peaks spreading faster: as workers are delayed in their copy of the pivot row by contention, more workers become ready for the current pivot row and join in the contention to acquire a copy.

Analysis with our toolkit and development of equations that model the imbalance in computation have provided us with a deep understanding of the behavior of our parallel algorithm for upper triangulation. From our analytic model, it is clear that the imbalance present in our improved program is a characteristic of any static partitioning of the problem. Only a dynamic partitioning strategy (see [Crowther *et al.*, 1985] which discusses a dynamic load balancing strategy for Gaussian elimination) offers a chance to eliminate some of this imbalance. Armed with this information we can make intelligent choices about how to proceed if the imbalance inherent in the algorithm causes unacceptable performance degradation as the program is applied to larger problem sizes or is executed on a larger number of processors.

7.3 General Lessons

In this chapter we have shown that the proposed model of execution tracing provides rich information for debugging and performance analysis, and furthermore, that our prototype toolkit facilitates analysis of the information contained in these traces. Our analysis case studies demonstrate a top-down methodology for debugging and performance analysis of parallel program executions using execution history traces. In both of the case studies, we began with examination of a high-level, abstract presentation of execution behavior using a graphical representation of execution history graphs. This representation facilitates understanding of both the logical and temporal relationships between the processes in an execution by displaying the flow of information that occurs through accesses to shared objects. Such an understanding guides analysis and focuses attention on the aspects of the program behavior that need to be considered both for debugging and in the development of performance models that accurately account for experimental data. With the visual representation of the execution history graphs as our guide, results of more detailed performance analyses can be understood in context.

In section 7.1, the analysis of the sorting program, we demonstrated how visual examination of execution history graphs provides valuable information for debugging by making the structure of communication evident. Also, we showed that simple measurements performed on the execution history graphs can be very useful in building performance models of real programs.

As an analysis example of a larger program execution, one with non-trivial execution time, the case study of the Gaussian elimination programs shows the importance of a multiplicity of views for understanding parallel program executions. To understand the complex relationships between the processes during a program execution, we needed to examine different aspects of the execution at a variety of scales. Furthermore, the Gaussian elimination analysis examples demonstrate the necessity for using graphical presentation methods to facilitate comprehension of performance data. The execution traces for the programs shown in the figures in the previous section contain hundreds (thousands in the case of the perspective plots) of elements. Textual presentation of data involving more than a handful of numbers makes it difficult to spot complex relationships. Thus, for analysis of large-scale parallel programs we resort to graphical means of examining the content of the execution traces.

Although the communication in the both merge sort and Gaussian elimination is highly structured, it is our contention that a similar amount of communication structure is present in most large-scale parallel applications that exploit data parallelism. Large-scale programs are typically constructed using an single-program-multiple-data (SPMD) model of computation which introduces regular periodic structure (and often symmetry, as in the case of the merge sort program) that can be exploited for both debugging and performance analysis.

In the Gaussian elimination analysis example presented in this chapter we use our toolkit to study a program execution consisting of 37 processes. We have been unable to trace an execution with more processes due to limitations in the network software configuration of our parallel processor.[9] While we found our toolkit adequate for analyzing executions this size, the memory requirements for our analysis toolkit were approaching the 24MB physical memory size of our largest workstation. Analyzing execution history graphs of size greater than the physical memory of the workstation used for the analysis causes the toolkit to thrash wildly, making analysis unbearably slow. Although this appears a severe limitation for scaling the toolkit to analyze larger program executions, we anticipate that reference locality can be improved in the program by using more principled allocation of graph nodes and bucket entries for hash tables.

[9]Recall that a network stream is needed for each process in an execution to record execution trace data.

8 Conclusions

The goal of this dissertation was to develop techniques that support a top-down methodology for debugging and analysis of large-scale parallel program executions on shared-memory multiprocessors. Pursuing this goal, we developed a formal model of communication in parallel program executions, a monitoring implementation based on this formal model, and an analysis toolkit that enables programmers to progress from an abstract graphical view of an entire parallel computation to fine-grain detail for debugging and analysis.

The techniques presented in this dissertation support a general approach to parallel program analysis. In order to effectively debug parallel programs, top-down analysis is essential. Since it is impractical to record every detail of a computation during execution, top-down analysis techniques are based on repeated analysis of multiple executions. To address the problem of repeating equivalent executions of parallel programs in highly parallel systems, a new scalable approach was necessary.

8.1 Contributions

As the basis for a new approach, we developed a formal model of parallel program executions based on shared-object communication. While such a model is clearly applicable for describing communication in shared-memory multiprocessors, it can also be applied in distributed systems by viewing message ports as shared objects. Using our model, we defined a precise notion of parallel program execution indistinguishability that is appropriate for debugging, and proved necessary and sufficient conditions for executions to be indistinguishable under our model. This model serves as a formal basis for cyclic debugging techniques that involve repeated examination of a program execution.

Based on this model, we developed a technique for recording minimal synchronization traces of parallel program executions. Each trace characterizes an instance of a program's execution behavior and enables deterministic replay of indistinguishable executions. Since our technique records only synchronization information, it scales better than message-logging approaches to systems with high communication rates. Unlike previous synchronization tracing techniques that support execution replay, our monitoring technique is fully parallel; therefore, it is scalable to highly parallel systems.

Finally, our execution tracing technique is independent of the particular form of interprocess communication used. This is important since shared-memory multiprocessors can support a wide variety of communication and synchronization abstractions.

One failing of previous approaches for replaying program executions is their inability to replay executions of programs that react to asynchronous events such as timeouts, asynchronous arrival of input, or asynchronous notification of exceptional conditions. To address this problem, we developed a software instruction counter technique for pinpointing the occurrence of asynchronous events. Tracing the occurrence of such events with our technique enables execution replay of programs (both sequential and parallel) that react to asynchronous events. Such a capability was never before available. This tracing technique is usable in systems with large-scale parallelism (tracing for individual processes is completely decoupled) and can easily be combined with our synchronization tracing technique.

Use of our execution tracing techniques enables deterministic replay of parallel program executions, which provides a solid foundation for debugging. With our techniques, cyclic debugging, which has been used so effectively in for sequential programs, is now possible for parallel programs, even those that react to asynchronous events or exhibit otherwise non-deterministic behavior. Being able to repeatedly examine indistinguishable executions of an erroneous program enables errors to be isolated reliably. Without the ability to replay erroneous executions on demand, debugging is largely based on luck; if an observed error cannot be readily reproduced, it may be difficult—if not impossible—to make forward progress towards diagnosing the underlying program fault.

Even with deterministic execution replay, tracking down the cause of an error may be difficult. Typically, the most difficult errors to pinpoint are those involving incorrect strategies for interprocess communication. Alone, traditional symbolic debugging tools are ill-suited to the task of diagnosing these types of errors. For large-scale parallel programs, trying to infer the dynamic relationships between processes by using symbolic debugging tools to examine process states can be cumbersome and confusing. To address this problem, we have developed a technique for top-down, graphics-based analysis of the patterns of interprocess communication that occur during an execution.

To provide a global perspective of the interprocess communication during a program execution, we use the information recorded in our synchronization traces (which have been annotated with timestamps and other identifying information for this purpose) to construct an execution history graph that details process interactions that occur through operations on shared objects. We have developed a graphical representation of these execution histories as space-time diagrams; this representation facilitates understanding of the dynamic relationships between processes. Our space-time diagrams provide an application independent view of interprocess communication and present not only information about the logical relationships between processes, which makes them useful for debugging, but also information about the temporal relationships between processes, which makes them useful for performance analysis. Temporal information available from our space-time diagrams includes process idle time while waiting for access to shared resources, the duration of intervals between synchronization points, and the relative synchrony/asynchrony of activities during an execution. Unlike animation

techniques, which display temporal information temporally using frame-based granularity, our space-time diagrams display temporal information spacially, enabling an entire execution to be surveyed at a glance. Since large-scale parallel programs typically use regular patterns of communication, irregularities in the communication patterns that might indicate program errors or performance anomalies are easy to spot.

Execution history graphs based on synchronization traces form the cornerstone for our integrated toolkit for debugging and performance analysis. We have described the design of a toolkit that integrates symbolic debugging tools that enable examination of program executions during controlled replay (based on a recorded synchronization trace), a graphical browser that enables visual inspection of execution histories and serves as a user-interface controlling symbolic debugging of execution replays, and a programmable interface for automating repetitive analysis tasks. In the sample analyses performed with a prototype implementation of our toolkit, we demonstrated that the integration of facilities for debugging and performance analysis supports effective development of parallel programs beginning with the debugging of an initial program implementation, and on through performance analysis and program tuning.

Our space-time diagrams have proven useful for diagnosing the causes of logical errors in the patterns of interprocess communication during program executions. Although space-time diagrams completely describe the aspects of program performance related to interprocess communication and provide a convenient abstract presentation of this information, the size of the diagrams for large-scale executions and the amount of detail they present makes them unsuitable for detailed analysis of global properties and trends. For detailed performance analysis, we have found our programmable interface indispensable for automating collection of information about particular aspects of a program's performance from its execution history graphs. We have shown through two case studies how both space-time diagrams and our programmable interface facilitate a top-down approach to debugging and performance analysis.

In conclusion, the tools and techniques developed as part of this research support a style of top-down analysis that begins with an abstract graphical view of a program execution and progresses to the examination of fine-grain detail necessary for symbolic debugging or detailed performance analysis. Our synchronization tracing techniques record compact characterizations of program executions based on partial orders of accesses to shared objects. For debugging, these traces enable execution replay that supports cyclic debugging. For performance analysis, these traces support a top-down approach that begins with an abstract view of the temporal relationships between processes as they communicate throughout a program execution, and progresses to investigation of specific aspects of the communication and synchronization behavior of the program under study. Analysis of traces, and execution replays based on these traces, is interactive, enabling programmers to focus the analysis. Analyses are repeatable, since the traces form a permanent record of the program execution. Finally, the tools based on these traces support an extensible set of analyses, providing an environment for development of special-purpose routines to analyze execution traces or control an execution replay. Future tools for debugging and performance analysis of large-scale parallel programs cannot be successful unless they provide similar capabilities.

8.2 Future Directions

8.2.1 Debugging

The most promising avenue of research for creating practical environments for debugging parallel programs involves a combination of static analysis, symbolic execution, on-the-fly detection of parallel access anomalies, and synchronization tracing techniques.

Alone, static analysis techniques for parallel programs require search of an exponentially large program state space. Young and Taylor [Young and Taylor, 1988] have shown that static analysis and symbolic execution can effectively be combined to reduce the search necessary. Their techniques use symbolic execution to prune unexecutable paths from the search space for static analysis, and static analysis to select paths for symbolic execution. However, even this combined approach cannot always conclusively determine the absence or presence of a potential parallel access anomaly (*e.g.*, in cases involving dynamically identified objects such as array elements).

In such cases, programs could be instrumented to detect parallel access anomalies during execution using techniques similar to those recently developed by Schonberg [Schonberg, 1989]. Schonberg's techniques compare variable sets accessed by blocks of code that execute concurrently. If the variable sets for two concurrent blocks contain conflicting accesses to the same variable, an error is signalled. Use of such an on-the-fly technique would enable detection of parallel access anomalies not caught using static analysis techniques.

While static analysis, symbolic execution, and on-the-fly error detection aid programmers in discovering stylized classes of programming errors, they provide no help in discovering semantic errors that cause a program's behavior to differ from its programmer's expectations. For this purpose, synchronization traces that enable repeatable analysis are essential to support a top-down approach to debugging. However, synchronization tracing methods for program replay are not alone sufficient for debugging. The technique for program replay presented in this dissertation works only if programs correctly use instrumented synchronization primitives to coordinate access to all shared data objects that may be the target of overlapping, conflicting operations (*i.e.*, no parallel access anomalies exist). If some process fails to use the appropriate protocol before accessing a shared object, the process may see the object in some inconsistent state. In such cases, deterministic execution replay cannot be guaranteed since, in general, it will be impossible to ensure that the shared object states seen by each process during execution replay are identical to those seen in the original execution. Thus, methods that detect parallel access anomalies are needed to supplement synchronization tracing methods for execution replay.

A second approach to combining synchronization tracing techniques with static analysis is to use a synchronization trace of an anomalous execution to focus the search for parallel access anomalies. If we wish to use static analysis to search for a parallel access anomaly that may be the cause of erroneous behavior observed in a particular execution (as defined by a synchronization trace), we can use the synchronization trace to prune

the search so that it only considers the portions of the state space that may have been visited during the execution.

8.2.2 Performance Analysis

The techniques used for analyzing program performance in this dissertation focus on the impact of interprocess communication, resource contention, and idle time. Such analysis provides insight into the effectiveness of strategies for problem partitioning and process coordination. Overall program performance, however, depends not only on these factors, but also on the efficiency of the sequential computation performed by each process. Determining the efficiency of sequential computation performed by individual processes was not addressed in this dissertation. Miller and Yang have developed techniques which focus on this aspect of performance tuning [Miller and Yang, 1987]. Their tools for profiling parallel program executions report a profile of the sequential computation along a program's critical path. Such profiles are generated by tracing procedure entries and exits between interprocess synchronization/communication events.

Miller and Yang's techniques seem adequate for tuning sequential computation occurring in single-program-multiple-data (SPMD) programs because reducing computation along the critical path will reduce the computation by each process (since all processes are identical), which should improve the program performance as a whole. However, for tuning parallel programs that are a collection of functionally distinct processes, profiles of only the activity along the critical path may not provide the right type of help. While Miller and Yang's techniques report profiles of the execution along the critical path and direct the programmer toward the most costly entries in this profile, this may not be the most expedient method of improving the performance of a parallel program. Other paths through the execution may be near the length of the critical path; thus, extensively tuning the most costly thing on the critical path may result in only a negligible decrease in overall execution time before other near-critical paths come into play. To address this problem, it seems that all paths need to be considered simultaneously using a dynamic programming algorithm. Guidance on which parts of the program should be improved should include the maximum improvement that can be gained by tuning any particular part, as well as the rate of improvement that tuning that part will provide.

8.2.3 Improving the Integrated Toolkit

In retrospect, many of the difficult implementation issues that arose during development of the integrated toolkit resulted from our hardware configuration with a "back-end" parallel machine. An issue that shaped the design of the toolkit was the placement of the user interface of the toolkit on a workstation (with single processor) rather than on our parallel machine itself. This organization causes some problems if we locate all our support for symbolic debugging in the user interface. If conditional breakpoints are set in multiple processes, execution replay would slow to a crawl as conditions need to be evaluated for multiple processes: our single process user interface would need to evaluate

the conditions sequentially. To avoid this bottleneck, we designed programmable debugger stubs that would execute on the target machine handling evaluation of conditional breakpoints (among other things) in parallel—one debugger stub would be available on each processor of our parallel machine. With these programmable stubs, the user interface would only become notified when a process was actually halted (when a conditional breakpoint was satisfied), rather than each time the condition needed to be evaluated. In hindsight, it would be better to simply run the user interface on the parallel machine, incorporating parallelism in the portion of the interface that deals with the target program. When this project began, constructing the entire toolkit to run directly on our parallel machine was not an option since it lacked secondary storage and the operating system configuration was not particularly stable. Now, as more mature operating system environments are becoming available for parallel machines, it seems prudent to re-implement the toolkit to run directly on the parallel machine.

In the previous chapter, it was mentioned that the current implementation of the toolkit was taxing the memory resources available on our largest workstation during analysis of very large executions. The toolkit apparently has poor locality of reference when constructing and analyzing large executions. This is evidenced by severe thrashing behavior when running the toolkit on machines that lack sufficient physical memory for the toolkit and the execution history under study. Since our goal was to be able to analyze very large executions, this problem needs to be addressed. If the toolkit is re-implemented to run directly on an available parallel machine, the parallel machine will almost certainly have more physical memory available, offering a simple solution to our problem.[1] A problem that to be investigated is how to improve the locality of reference during construction and analysis of our execution history graphs. Along these lines, since our implementation makes frequent use of hash tables in managing execution history graphs, it seems that we could greatly improve locality of reference in these tables by preallocating pools of memory for bucket chains rather than interspersing allocation of bucket entries with allocation of nodes in the execution history graph.

[1] A parallel implementation of the toolkit would also help with the computational requirements for construction and analysis of these large graphs.

Bibliography

[ACM, 1979] "ACM Forum: Comments on Social Processes and Proofs," *Communications of the ACM*, 22(11):621–630, November 1979.

[Allen and Padua, 1987] Todd R. Allen and David A. Padua, "Debugging Fortran on a Shared Memory Machine," In *Proc. of the 1987 International Conference on Parallel Processing*, pages 721–727, August 1987.

[Am29000, 1988] Advanced Micro Devices, Sunnyvale, CA, *Am29000 32-bit Streamlined Instruction Processor Users Manual*, 1988.

[Appelbe and McDowell, 1988a] William F. Appelbe and Charles E. McDowell, "Developing Multitasking Applications Programs," In *Proc. of the 21st Annual Hawaii International Conference on System Sciences*, volume 2, pages 94–101, Kailua-Kona, HI, January 1988.

[Appelbe and McDowell, 1988b] William F. Appelbe and Charles E. McDowell, "Integrated Tools for Debugging and Developing Multitasking Programs," In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 78–88, Madison, WI, May 1988, Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[Babich, 1979] Alan F. Babich, "Proving Total Correctness of Parallel Programs," *IEEE Transactions on Software Engineering*, SE-5(6):558–574, November 1979.

[Baiardi *et al.*, 1986] F. Baiardi, N. DeFrancesco, and G. Vaglini, "Development of a Debugger for a Concurrent Language," *IEEE Transactions on Software Engineering*, SE-12(4):547–553, April 1986.

[Balzer, 1969] R.M. Balzer, "EXDAMS — EXtensible Debugging and Monitoring System," *AFIPS Spring Joint Computer Conference*, pages 567–580, 1969.

[Bates and Wileden, 1983] Peter Bates and Jack Wileden, "High Level Debugging of Distributed Systems: The Behavioral Abstraction Approach," Technical Report COINS 83-29, Computer and Information Sciences, University of Massachusetts, 1983.

[BBN Laboratories, 1986] BBN Laboratories, "Butterfly Parallel Processor Overview," Technical Report 6148, Version 1, BBN Laboratories, Cambridge, MA, March 1986.

[BBN Laboratories, 1987] BBN Laboratories, *Chrysalis Programmer's Manual*, BBN Laboratories, Cambridge, MA, May 1987.

[Becker and Chambers, 1984] Richard A. Becker and John M. Chambers, *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth Advanced Book Program, Belmont, CA, 1984.

[Brantley *et al.*] William C. Brantley, Kevin P. McAuliffe, and Ton A. Ngo, "RP3 Performance Monitoring Hardware".

[Bruegge and Hibbard, 1983] Bernd Bruegge and Peter Hibbard, "Generalized Path Expressions: A High Level Debugging Mechanism," In *Proc. ACM Software Engineering Symp. on High-Level Debugging*, pages 34–44, Pacific Grove, CA, March 1983.

[Callahan and Subhlok, 1988] David Callahan and Jaspal Subhlok, "Static Analysis of Low-Level Synchronization," In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, Madison, WI, May 1988, Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[Cargill and Locanthi, 1987] T.A. Cargill and B.N. Locanthi, "Cheap Hardware Support for Software Debugging and Profiling," In *Proc. of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82–83, Palo Alto, CA, October 1987.

[Carver and Tai, 1986] R. Carver and K.-C. Tai, "Reproducible Testing of Concurrent Programs Based on Shared Variables," *Proc. 6th International Conference on Distributed Computing Systems*, pages 428–433, May 1986.

[Chandy and Lamport, 1985] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[Chiu, 1984] S.Y. Chiu, "Debugging Distributed Computations in a Nested Atomic Action System," Technical Report MIT/LCS/TR-327, Massachusetts Institute of Technology, Laboratory for Computer Science, December 1984, PhD thesis.

[Cooper, 1987] Robert Cooper, "Pilgrim: A Debugger for Distributed Systems," In *Proc. 7th International Conference on Distributed Computing Systems*, pages 21–25, Berlin, W. Germany, September 1987.

[Courtois *et al.*, 1971] P.J. Courtois, F. Heymans, and D.L. Parnas, "Concurrent Control with Readers and Writers," *Communications of the ACM*, 14(10):667–668, October 1971.

[Crowther *et al.*, 1985] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar, "Performance Measurements on a 128-node Butterfly Parallel Processor," In *Proc. of the 1985 International Conference on Parallel Processing*, pages 531–540, St. Charles, Illinois, August 1985.

[Curtis and Wittie, 1982] R.S. Curtis and L.D. Wittie, "BugNet: A Debugging System for Parallel Programming Environments," In *Proc. 3rd International Conference on Distributed Computing Systems*, pages 394–399, Miami, FL, October 1982.

[De Millo *et al.*, 1979] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis, "Social Processes and Proofs of Theorems and Programs," *Communications ACM*, 22(5):271–280, May 1979.

[Digital Equipment Corporation, 1981] Digital Equipment Corporation, *VAX Architecture Handbook*, Digital Equipment Corporation, Maynard, MA, 1981.

[Dijkstra, 1968] E.W. Dijkstra, "The Structure of the 'THE' Multiprogramming System," *Communications ACM*, 11(5):341–346, May 1968.

[DiMaio *et al.*, 1985] A. DiMaio, S. Ceri, and S.C. Reghizzi, "Execution Monitoring and Debugging Tool for Ada Using Relational Algebra," In *Proc. of the Ada International Conference*, pages 109–123, Paris, France, May 1985.

[Duda *et al.*, 1987] A. Duda, G. Harrus, Y. Haddad, and G. Bernard, "Estimating Global Time in Distributed Systems," In *Proc. 7th International Conference on Distributed Computing Systems*, pages 299–306, Berlin, West Germany, September 1987.

[Elshoff, 1988] I. J. P. Elshoff, "A Distributed Debugger for Amoeba," In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 1–10, Madison, WI, May 1988, Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[Emrath and Padua, 1988] Perry A. Emrath and David A. Padua, "Automatic Detection of Nondeterminacy in Parallel Programs," In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, Madison, WI, May 1988, Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[Encore, 1987] "Multimax Technical Summary," Technical report, March 1987.

[Fairchild Semiconductor Corporation, 1987] Fairchild Semiconductor Corporation, *CLIPPER 32-bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Fetzer, 1988] James H. Fetzer, "Program Verification: The Very Idea," *Communications of the ACM*, 31(9):1048–1063, September 1988.

[Flynn, 1972] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[Fowler *et al.*, 1988] R.J. Fowler, T.J. LeBlanc, and J.M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors," In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173, Madison, WI, May 1988, Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[Gait, 1985] J. Gait, "A Debugger for Concurrent Programs," *Software—Practice and Experience*, 15(6):539–554, June 1985.

[Garcia and Berman, 1985] M. Garcia and W. Berman, "An Approach to Concurrent Systems Debugging," In *Proc. 5th International Conference on Distributed Computing Systems*, pages 507–514, Denver, CO, May 1985.

[Garcia-Molina *et al.*, 1984] H. Garcia-Molina, F. Germano, and W.H. Kohler, "Debugging a Distributed Computing System," *IEEE Transactions on Software Engineering*, SE-10(2):210–219, March 1984.

[Gardner, 1988] Robert B. Gardner, "SPARC Scalable Processor Architecture," Sun *Technology*, 1(3):42–55, 1988.

[Gottlieb *et al.*, 1983] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir, "The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[Gottlieb and Kruskal, 1981] Allan Gottlieb and Clyde P. Kruskal, "Coordinating Parallel Processors: A Partial Unification," *Computer Architecture News*, 9(6):16–24, October 1981.

[Graham *et al.*, 1982] Susan L. Graham, Paeter B. Kessler, and Marshall K. McKusick, "gprof: A Call Graph Execution Profiler," In *Proc. of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126. SIGPLAN notices, 17(6), June 1982.

[Gustafson, 1988] John L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, 31(5):532–533, May 1988.

[Harter *et al.*, 1985] Paul K. Harter, Dennis M. Heimbigner, and Roger King, "IDD: An Interactive Distributed Debugger," In *Proc. 5th International Conference on Distributed Computing Systems*, pages 498–506, Denver, CO, May 1985.

[Helmbold and Luckham, 1984] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," In *IEEE Computer Society Conference on Ada Applications and Environments*, pages 97–105, St. Paul, MN, October 1984.

[Hewlett-Packard, 1987] Hewlett-Packard Company, *Precision Architecture and Instruction Reference Manual*, second edition, June 1987.

[Hillis, 1985] Daniel W. Hillis, *The Connection Machine*, MIT Press, 1985.

[Hough and Cuny, 1987] Alfred A. Hough and Janice E. Cuny, "Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation," In *Proc. of the 1987 International Conference on Parallel Processing*, pages 735–738, August 1987.

[Joyce *et al.*, 1987] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger, "Monitoring Distributed Systems," *ACM Transactions on Computer Systems*, 5(2):124–150, May 1987.

[Karp, 1987] Alan H. Karp, "Programming for Parallelism," *Computer*, 20(5):43–57, May 1987.

[Knuth, 1973] Donald E. Knuth, *The Art of Computer Programming: Volume 3 Sorting and Searching*, Addison Wesley, Reading, MA, 1973.

[Lamport, 1985] L. Lamport, "On Interprocess Communication," Technical report, Digital Equipment Corporation's Western Research Lab, December 1985.

[Lamport, 1977] Leslie Lamport, "Proving the Correctness of Multiprocess Programs," *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[Lamport, 1978] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, 21(7):558–565, July 1978.

[Lauesen, 1979] S. Lauesen, "Debugging Techniques," *Software—Practice and Experience*, 9:51–63, January 1979.

[LeBlanc and Robbins, 1985] R.J. LeBlanc and A.D. Robbins, "Event Driven Monitoring of Distributed Programs," *Proc. 5th International Conference on Distributed Computing Systems*, pages 515–522, May 1985.

[LeBlanc *et al.*, 1986] Thomas J. LeBlanc, Neal M. Gafter, and Takahide Ohkami, "SMP: A Message-Based Programming Environment for the BBN Butterfly," Butterfly Project Report 8, Department of Computer Science, University of Rochester, July 1986.

[LeBlanc and Jain, 1987] Thomas J. LeBlanc and Sanjay Jain, "Crowd Control: Coordinating Processes in Parallel," In *Proc. of the 1987 International Conference on Parallel Processing*, pages 81–84, St. Charles, IL, August 1987.

[LeBlanc and Mellor-Crummey, 1987] Thomas J. LeBlanc and John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[LeBlanc, 1986] T.J. LeBlanc, "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study," In *Proc. of the 1986 International Conference on Parallel Processing*, pages 463–466, St. Charles, Illinois, August 1986.

[LeBlanc, 1988] T.J. LeBlanc, "Structured Message Passing on a Shared-Memory Multiprocessor," In *Proc. of the 21st Annual Hawaii Conference on System Sciences*, January 1988.

[LeDoux and Parker, 1985] C.H. LeDoux and D.S. Parker, "Saving Ada Traces for Debugging," In *Proc. of the Ada International Conference*, pages 97–107, Paris, France, May 1985.

[Lin and LeBlanc, 1988] Chu-Chung Lin and Richard J. LeBlanc, "Event-based Debugging of Object/Action Programs," In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 23–34, Madison, WI, May 1988, Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[Linton, 1983] M.A. Linton, "Queries and Views of Programs Using a Relational Database," Technical Report UCB/CSD/83/164, Computer Science Division (EECS), University of California, Berkeley, December 1983, PhD thesis.

[Lockyer, 1964] K. G. Lockyer, *Introduction to Critical Path Analysis*, Pitman Publishing Co., New York, NY, 1964.

[MC88100, 1988] Motorola, Inc., *MC88100 RISC Microprocessor User's Manual*, 1988.

[McDaniel, 1977] Gene McDaniel, "Metric: a Kernel Instrumentation System for Distributed Environments," *Proc. of the 6th ACM Symposium on Operating System Principles*, pages 93–99, November 1977.

[Mellor-Crummey, 1987] John M. Mellor-Crummey, "Concurrent Queues: Practical Fetch-and-Φ Algorithms," Technical Report 229, Department of Computer Science, University of Rochester, November 1987.

[Mellor-Crummey and LeBlanc, 1989] John M. Mellor-Crummey and Thomas J. LeBlanc, "A Software Instruction Counter," In *Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78–86, Boston, MA, April 1989.

[Miller *et al.*, 1986] B. P. Miller, C. Macrander, and S. Sechrest, "A Distributed Programs Monitor for Berkeley Unix," *Software—Practice and Experience*, 16(2):183–200, February 1986.

[Miller, 1985a] B.P. Miller, "Parallelism in Distributed Programs: Measurement and Prediction," Technical report, Department of Computer Science, University of Wisconsin at Madison, May 1985.

[Miller, 1985b] B.P. Miller, "Performance Characterization of Distributed Programs," Technical Report UCB/CSD/85/197, Computer Science Division, EECS, University of California, Berkeley, 1985.

[Miller, 1988] B.P. Miller, "DPM: A Measurement System for Distributed Programs," *IEEE Transactions on Computers*, 37(2):243–248, February 1988.

[Miller and Choi, 1986] B.P. Miller and J.D. Choi, "Breakpoints and Halting in Distributed Programs," Technical report, Department of Computer Science, University of Wisconsin at Madison, July 1986.

[Miller and Choi, 1988] B.P. Miller and J.D. Choi, "A Mechanism for Efficient Debugging of Parallel Programs," In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 141–150, Madison, WI, May 1988, Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[Miller and Yang, 1987] B.P. Miller and C-Q. Yang, "IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs," In *Proc. 7th International Conference on Distributed Computing Systems*, pages 482–489, Berlin, West Germany, September 1987.

[Motorola, 1985] Motorola, *68020 32-bit Microprocessor User's Manual, Second Edition*, Prentice Hall, Englewood Cliffs, NJ, 1985.

[Moussouris *et al.*, 1986] J. Moussouris, L. Crudele, D. Freitas, C. Hansen, E. Hudson, R. March, S. Przybylski, T. Riordan, C. Rowen, and D. Van't Hof, "A CMOS RISC Processor with Integrated System Functions," In *Proc. of the 1986 COMPCON.* IEEE, March 1986.

[Muchnick, 1988] Steven S. Muchnick, "Optimizing Compilers for SPARC," Sun *Technology*, 1(3):64–77, 1988.

[Osterweil, 1981] Leon Osterweil, *Using Data Flow Tools in Software Engineering*, chapter 8, pages 237–263, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[Owicki and Gries, 1976] Susan Owicki and David Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Communications of the ACM*, 19(5):279–285, May 1976.

[Padua and Wolfe, 1986] David A. Padua and Michael J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM*, 29(12):1184–1201, December 1986.

[Pan and Linton, 1988] Douglas Z. Pan and Mark A. Linton, "Supporting Reverse Execution of Parallel Programs," In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, Madison, WI, May 1988, Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[Panangaden and Taylor, 1988] Prakash Panangaden and Kim Taylor, "Concurrent Common Knowledge: A New Definition of Agreement for Asynchronous Systems," In *Proc. of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 197–209, Toronto, Ontario, Canada, August 1988.

[Peterson, 1983] Gary L. Peterson, "Concurrent Reading While Writing," *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[Pfister *et al.*, 1985] G.F. Pfister *et al.*, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," In *Proc. of the 1985 International Conference on Parallel Processing*, pages 764–771, St. Charles, Illinois, August 1985.

[Powell and Presotto, 1983] M.L. Powell and D.L. Presotto, "PUBLISHING: A Reliable Broadcast Communication Mechanism," *Operating Systems Review*, 17(5):100–109, 1983.

[Redell, 1988] David D. Redell, "Experience with Topaz TeleDebugging," In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 35–44, Madison, WI, May 1988, Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[Reed and Kanodia, 1979] David P. Reed and Rajendra K. Kanodia, "Synchronization with Eventcounts and Sequencers," *Communications of the ACM*, 22(2):115–123, 1979.

[Richardson, 1988] Rick Richardson, "Dhrystone 2.1 Benchmark," Usenet Distribution, December 1988.

[Rosen, 1976] Barry K. Rosen, "Correctness of Parallel Programs: The Church-Rosser Approach," *Theoretical Computer Science*, 2:183–207, 1976.

[Rubin *et al.*, 1988] Robert V. Rubin, Larry Rudolph, and Dror Zernik, "Debugging Parallel Programs in Parallel," In *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 216–225, Madison, WI, May 1988, Special issue of SIGPLAN Notices, 24(1), Jan. 1989.

[Scheifler and Gettys, 1986] Robet W. Scheifler and Jim Gettys, "The X Window System," *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[Schiffenbauer, 1981] R.D. Schiffenbauer, "Interactive Debugging in a Distributed Computational Environment," Technical Report MIT/LCS/TR-264, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1981, Master's Thesis.

[Schonberg, 1989] Edith Schonberg, "On-The-Fly Detection of Acess Anomalies," In *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 285–297, June 1989.

[Scott, 1986a] Michael L. Scott, "The Interface Between Distributed Operating System and High-Level Programming Language," In *Proc. of the 1986 International Conference on Parallel Processing*, pages 242–249, August 1986.

[Scott, 1986b] Michael L. Scott, "LYNX Reference Manual," Butterfly Project Report 7, Department of Computer Science, University of Rochester, March 1986.

[Segall and Rudolph, 1985] Z. Segall and L. Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software*, 2(6):22–37, November 1985.

[Seitz, 1985] Charles L. Seitz, "The Cosmic Cube," *Communications of the ACM*, 28(1):22–33, January 1985.

[Sequent, 1984] Sequent Computer Systems, Inc., *Balance 8000 System Technical Summary*, 1984.

[Smith, 1984] Edward T. Smith, "Debugging Tools for Message-Based, Communicating Processes," In *Proc. 4th International Conference on Distributed Computing Systems*, pages 303–310, San Francisco, CA, May 1984.

[Snodgrass, 1982] R. Snodgrass, "Monitoring Distributed Programs: A Relational Approach," Technical report, Department of Computer Science, Carnegie-Mellon University, December 1982, PhD thesis.

[Snodgrass, 1988] Richard Snodgrass, "A Relational Approach to Monitoring Complex Systems," *ACM Transactions on Computer Systems*, pages 157–196, May 1988.

[Strang, 1980] Gilbert Strang, *Linear Algebra and Its Applications*, Academic Press, Inc., Orlando, FL, second edition edition, 1980.

[Taylor, 1983a] Richard N. Taylor, "Complexity of Analyzing the Synchronization Structure of Concurrent Programs," *Acta Informatica*, 19:57–84, 1983.

[Taylor, 1983b] Richard N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs," *Communications of the ACM*, 26(5):362–376, May 1983.

[Taylor and Osterweil, 1980] Richard N. Taylor and Leon J. Osterweil, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," *IEEE Transactions on Software Engineering*, SE-6(3):265–277, May 1980.

[Ullman, 1984] Jeffrey D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Rockville, MD, 1984.

[Vitanyi and Awerbuch, 1986] P. Vitanyi and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware," *Proc. 27th Annual Symp. on Foundations of Computer Science*, October 1986.

[Weber, 1983] J.C. Weber, "Interactive Debugging of Concurrent Programs," In *Proc. ACM Software Engineering Symp. on High-Level Debugging*, pages 112–113, Pacific Grove, CA, March 1983.

[Weicker, 1988] Reinhold P. Weicker, "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules," *SIGPLAN Notices*, pages 49–62, August 1988.

[Yang and Miller, 1988] Cui-Qing Yang and Barton P. Miller, "Critical Path Analysis for the Execution of Parallel and Distributed Programs," In *Proc. 8th International Conference on Distributed Computing Systems*, pages 366–373, San Jose, CA, June 1988.

[Yew *et al.*, 1987] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[Young and Taylor, 1988] Michal Young and Richard N. Taylor, "Combining Static Concurrency Analysis with Symbolic Execution," *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.

[Yuasa and Hagiya, 1985] Taiichi Yuasa and Masami Hagiya, "Kyoto Common Lisp Report," Technical report, Research Institute for Mathematical Sciences, Kyoto University, 1985.

Appendix A

A CREW Lock Implementation Without a Critical Section

In this appendix, we present an implementation of a CREW lock that makes use of the atomic fetch-and-add primitive provided by the BBN Butterfly Parallel Processor[BBN Laboratories, 1986] to avoid the need for a critical section. In practice, the lack of a critical section substantially improves performance when there is a large number of readers simultaneously attempting access.

To avoid a critical section for readers, this CREW lock implementation uses an optimistic strategy. When a reader attempts to acquire a lock, it assumes no writer is active and increments the count of active readers. Next it checks to see if a writer is present. If not, the reader is granted access to the object. If a writer is active, the reader decrements the count of active readers and spins until the writer has finished. At this time, the reader repeats the same optimistic access sequence assuming that no writer is present.

```
type shared_object_header is record
  ticket_number: ATOMICshort;
  now_serving: short;
  write_active: boolean;
  version: short;
  active_rdrs, completed_rdrs: ATOMICshort;
end record;


ReaderEntry(var object: shared_object_header);
  if mode = MONITOR then
    loop
      AtomicAdd(object.active_rdrs, 1);
      if object.write_active = FALSE then exit loop;
      else
        AtomicAdd(object.active_rdrs, -1);
        while (object.write_active = TRUE) delay;
      end if;
    end loop;
    WriteHistoryTape(object.version);
  else
    // wait for appropriate version of the object
    my_version := ReadHistoryTape();
    while (my_version != object.version) delay;
  end if;
end ReaderEntry;


ReaderExit(var object: shared_object_header);
  AtomicAdd(object.completed_rdrs, 1);
  AtomicAdd(object.active_rdrs, -1);
end ReaderExit;
```

```
WriterEntry(var object: shared_object_header);
  if mode = MONITOR then
    waited := FALSE;
    my_ticket := AtomicAdd(object.ticket_number, 1);
    while (object.now_serving != my_ticket) delay;
    object.write_active := TRUE;
    // wait for active readers to finish
    while (object.active_rdrs > 0) do delay;
    WriteHistoryTape(object.version);
    WriteHistoryTape(object.completed_rdrs);
  else
    my_version := ReadHistoryTape();
    my_readers := ReadHistoryTape();
    // wait until readers done with prev. version of the object
    while ((my_version != object.version) or
        (my_readers != object.completed_rdrs)) do delay;
  end if;
end WriterEntry;


WriterExit(var object: shared_object_header);
  object.completed_rdrs := 0;   // zero rdr count for new version
  object.version++;             // advance version number
  object.write_active := FALSE; // allow readers to proceed
  object.now_serving++;         // allow next writer to proceed
end WriterExit;
```

Appendix B

A CREW Protocol that Generates Augmented Traces

In this appendix, we present the actual implementation of the optimistic CREW lock algorithm given in appendix A that we use to record augmented synchronization traces of programs executing on our BBN Butterfly Parallel Processor [BBN Laboratories, 1986]. These traces serve as the foundation for our prototype integrated toolkit described in chapter 6.

The trace information recorded here is considerably more detailed than that recorded by the algorithms in chapter 4. Instead of recording traces as a continuous stream of integer values (corresponding to version numbers and reader counts), here, traces are composed of a stream of records, each of which has a prefix containing an identifying type code and a unique object id. Also, trace entries contain real-time clock values which indicate the time a lock was requested, the time a lock was granted, and the time a lock was released. This additional information enables the traces to be used for analyzing program behavior and performance rather than just execution replay. The cost of logging the additional data is small and it makes the traces much more useful. These access protocols were written to trade code space for execution time. They were written to minimize the number of calls to the data logging routines since these routines do a bit of complicated calculation to determine where the data should be logged among a set of local buffers available to the process.

When a process attempts to acquire a lock it is imperative that the intent to acquire the lock is recorded even if access is not granted; otherwise, if deadlock occurs, there will be no evidence that a set of processes is waiting for locks; the traces will simply end. To minimize the number of calls to the data-logging routines, a record of the intent to access an object is not written out immediately upon the request. First an attempt is made to acquire the lock so the intent to acquire and the acquisition can be written out in one unit. If the lock is not immediately available, the record of intent is immediately written out before waiting and trying the lock again. When the lock is subsequently acquired, any remaining information that needs to be recorded is logged.

```
// shared_object: this is a C++ package that implements CREW
//     protocols for controlling access to shared objects
//
// GOALS: optimize with following criteria in mind
```

```
//    1) speed of normal execution case is more important than
//       replay case
//    2) lock collisions between readers and writers are expected
//       to be infrequent

// NOTE: the real-time clock value (32 bit value, 62.5 us ticks)
//       is read by reading the ''magic location'' named ''rtc''

typedef int (*signum_pred)(long);

class shared_object {
  ATOMICshort ticket_number;
  short now_serving;
  boolean write_active;
  short version_number;
  ATOMICshort active_rdrs;
  ATOMICshort completed_rdrs;
  inline void logtimestamp(EVENTTYPE);
public:
  OID object_id;
  sharedmemory();
  void read_start();
  void write_start();
  void read_end();
  void write_end();
  boolean poll_start(signum_pred p, long argument, int locktype);
};


// function logtimestamp: update the replaylog with a record
//   containing the id of the current object and the current time
inline void shared_object::logtimestamp(EVENTTYPE etype)
{
  access_end_event_type event;
  if (normal_execution) {
    // record timestamp for end of access
    event.type = etype;
    event.object_id = object_id;
    event.time = rtc;
    replaylog.write(&event,sizeof(event));
  }
  else {
    increment_event_counter();     // for ''event breakpoints''
    logread(&event,sizeof(event)); // read event from log
    checktype(event.type,etype);   // verify same typecode
```

```
  }
}


// constructor for to initialize the object header
shared_object::shared_object()
{
  ticket_number = 0;
  now_serving = 0;
  write_active = 0;
  version = 0;
  active_rdrs = 0;
  completed_rdrs = 0;
  object_id = (OID) makephys(this); // use physical addr as id
}


// function read_start: acquire a READ lock on a shared memory
//   object
void shared_object::read_start()
{
  read_start_event_type event;
  if (normal_execution) {

    // fill in access request time and type code
    event.prefix.request_time = rtc;
    event.prefix.type = MEMORY_READ_START;
    event.prefix.object_id = object_id;

    // flag to indicate that a that a spin lock was never used
    // and the entire read_start event can be logged as a unit
    boolean waited = FALSE;

    for(;;) {
      Atomic_increment(&active_rdrs);
      if (!write_active) {
        event.suffix.access_time = rtc;
        event.suffix.version = version;
        if (!waited) {
          // no spin lock ever used --> prefix of record never
          // logged, so log the entire record here
          replaylog.write(&event,sizeof(event));
        }
        else {
          // spin lock used --> prefix of record logged, so only
```

```
          // log suffix here
          replaylog.write(&event.suffix,sizeof(event.suffix));
        }
        break;
      }
      else {
        if (!waited) {
          // before spinning, log the attempt to access the
          // shared memory object
          replaylog.write(&event.prefix,sizeof(event.prefix));
          waited = TRUE;
        }
        Atomic_decrement(&active_rdrs);
        while (write_active) short_delay(SWITCH_SIZE);
      }
    }
  }
  else {
    increment_event_counter(); // for ''event breakpoints''
    logread(&event,sizeof(event));
    checktype(event.prefix.type,MEMORY_READ_START);
    // wait for appropriate version of the object
    while (event.suffix.version != version)
      short_delay(SWITCH_SIZE);
  }
}


// function read_end: release a READ lock on a shared memory
//   object
void shared_object::read_end()
{
  // release lock
  Atomic_increment(&completed_rdrs);
  Atomic_decrement(&active_rdrs);
  logtimestamp(MEMORY_READ_END);
}


// function write_start: acquire a WRITE lock on a shared memory
//   object
void shared_object::write_start()
{
  write_start_event_type event;
  if (normal_execution) {
```

```
    // fill in access request time and type code
    event.prefix.request_time = rtc;
    event.prefix.type = MEMORY_WRITE_START;
    event.prefix.object_id = object_id;

    // flag to indicate that a that a spin lock was never used
    // and the entire read_start event can be logged as a unit
    boolean waited = FALSE;

    int my_ticket = Atomic_increment(&ticket_number);
    while (now_serving != my_ticket) {
      if (!waited) {
        // before the first spin lock, log the attempt to access
        // the shared memory object
        replaylog.write(&event.prefix,sizeof(event.prefix));
        waited = TRUE;
      }
      else short_delay(SWITCH_SIZE);
    }

    write_active = TRUE;

    // wait for active readers to finish
    while (active_rdrs > 0) short_delay(SWITCH_SIZE);

    event.suffix.access_time = rtc;
    event.suffix.version = version;
    event.suffix.reader_count = completed_rdrs;

    if (!waited) {
      // no spin lock ever used --> prefix of record never
      // logged, so log the entire record here
      replaylog.write(&event,sizeof(event));
    }
    else {
      // spin lock used --> prefix of record logged, so only
      // log suffix here
      replaylog.write(&event.suffix,sizeof(event.suffix));
    }
  }
  else {
    increment_event_counter(); // for ''event breakpoints''
    logread(&event,sizeof(event));
    checktype(event.prefix.type,MEMORY_WRITE_START);
```

```
    // wait until readers done with prev. version of the object
    while ((event.suffix.version != version) ||
        (event.suffix.reader_count != completed_rdrs))
      short_delay(SWITCH_SIZE);
  }
}


// function write_end: release a WRITE lock on the shared
//   memory object
void shared_object::write_end()
{
  completed_rdrs = 0;   // zero reader count for new version
  version++;            // advance version number
  write_active = FALSE; // allow readers to proceed
  now_serving++;        // allow next writer to proceed
  logtimestamp(MEMORY_WRITE_END);
}


// function poll_start: poll the object with the supplied
//    predicate, when the predicate evaluates to TRUE, acquire
//    the requested type of lock READ, WRITE, or NONE
//
//    lock type NONE is used strictly for synchronizing with
//    some other process that is manipulating the object when no
//    operation on the object is needed
boolean shared_object::poll_start(
  signum_pred p,    // cond to be satisfied before acquiring lock
  long argument,    // argument to be passed to predicate p
  int locktype      // type of lock desired
)
{
  if (normal_execution) {
    poll_event_prefix_type event;
    event.request_time = rtc;
    event.object_id = object_id;

    switch(locktype)
    {
    case NONE:
      event.type = MEMORY_POLL_NONE;
      break;
    case READ:
      event.type = MEMORY_POLL_READ_START;
```

```
  break;
case WRITE:
  event.type = MEMORY_POLL_WRITE_START;
  break;
}
replaylog.write(&event,sizeof(event));
boolean notdone = TRUE;
while(notdone) {
  // wait until the predicate tests non-null
  while(!(*p)(argument))
    short_delay(SWITCH_SIZE);

  switch(locktype) {
  case NONE: {
    // acquire a read lock
    poll_event_suffix_type event;

    for(;;) {
      Atomic_increment(&active_rdrs);
      if (!write_active) break;
      else {
        Atomic_decrement(&active_rdrs);
        while (write_active) short_delay(SWITCH_SIZE);
      }
    }
    event.access_time = rtc;

    int value = (*p)(argument);

    event.version = version;
    event.index = value;

    switch(value) {
    case -1:
      Atomic_decrement(&active_rdrs);
      replaylog.write(&event,sizeof(event));
      return 0;
    case 0:
      Atomic_decrement(&active_rdrs);
      break;
    case 1: {
      Atomic_increment(&completed_rdrs);
      Atomic_decrement(&active_rdrs);
      replaylog.write(&event,sizeof(event));
      return 1;
```

```
    }
    }

    break;
  }
  case READ: {
    // acquire a read lock
    poll_event_suffix_type event;

    for(;;) {
      Atomic_increment(&active_rdrs);
      if (!write_active) break;
      else {
        Atomic_decrement(&active_rdrs);
        while (write_active) short_delay(SWITCH_SIZE);
      }
    }
    event.access_time = rtc;

    int value = (*p)(argument);

    event.version = version;
    event.index = value;

    switch(value) {
    case -1:
      Atomic_decrement(&active_rdrs);
      replaylog.write(&event,sizeof(event));
      return 0;
    case 0:
      Atomic_decrement(&active_rdrs);
      break;
    case 1: {
      replaylog.write(&event,sizeof(event));
      return 1;
    }
    }

    break;
  }
  case WRITE: {
    // acquire a write lock
    poll_write_event_suffix_type event;

    int my_ticket = Atomic_increment(&ticket_number);
```

```
        while (now_serving != my_ticket)
          short_delay(SWITCH_SIZE);
        write_active = TRUE;

        event.access_time = rtc;

        int value = (*p)(argument);

        event.version = version;
        event.index = value;

        switch(value) {
        case -1:
          write_active = FALSE;
          event.reader_count = 0;
          replaylog.write(&event,sizeof(event));
          return 0;
        case 0:
          write_active = FALSE;
          event.reader_count = completed_rdrs;
          break;
        case 1:
          // wait for active readers to finish
          while (active_rdrs > 0) short_delay(SWITCH_SIZE);
          event.reader_count = completed_rdrs;
          replaylog.write(&event,sizeof(event));
          return 1;
        }
      }
      }
    }
  } else {
    EVENTTYPE etype;
    poll_event_prefix_type event;

    increment_event_counter(); // for ''event breakpoints''
    logread(&event,sizeof(event));

    switch(locktype)
    {
    case READ:
      etype = MEMORY_POLL_READ_START;
      break;
    case NONE:
      etype = MEMORY_POLL_NONE;
```

```
      break;
    case WRITE:
      etype = MEMORY_POLL_WRITE_START;
      break;
    }

  // verify that the event read from the log is what we expect
    checktype(event.type,etype);

    switch(locktype)
    {
    case READ:
    case NONE: {
      poll_event_suffix_type event;
      logread(&event,sizeof(event));

      if (event.index == -1) return 0;

      while (event.version != version)
        short_delay(SWITCH_SIZE);

      if (locktype == READ) break;
      Atomic_increment(&completed_rdrs);
      break;
    }
    case WRITE: {
      poll_write_event_suffix_type event;
      logread(&event,sizeof(event));

      if (event.index == -1) return 0;

      while (event.version != version)
          short_delay(SWITCH_SIZE);

      while (event.reader_count != completed_rdrs)
        short_delay(SWITCH_SIZE);
      }
      break;
    }
    return 1;
  }
}
```

Appendix C

Lisp Code For Analysis of Gaussian Elimination

This appendix contains the Lisp code written to collect data from Gaussian Elimination execution history graphs. Each of the functions marked with asteriks was used to called to collect the data for a graph shown in Chapter 7; the other functions are auxiliary functions that support the data collection.

The macro s-event-p is a predicate which insures that returns t if its argument is an event. Using this predicate offers some measure of confidence that it is safe to pass the Lisp pointer into the C world which performs no checking. This function provides the only safety across the interface. If an invalid pointer is passed into the C world from Lisp, dereferencing it may cause the toolkit to crash. The macro s-history-p provides the same facility for testing whether a Lisp object represents an execution history data structure.

The functions event-last, event-next, event-stime, event-exit, event-opcode, event-value, and event-access provide access to some of the fields maintained for each node in an execution history graph. The function hist-number-procs returns the number of processes in an execution history.

The comments in the code are self-explanatory.

```
;;-----------------------------------------------------------------
;; event record typecode definitions
;;-----------------------------------------------------------------
(defconstant POLLREADSTART 0)
(defconstant READEND 16)
(defconstant WRITEEND 18)
(defconstant POLLWRITESTART 1)
(defconstant WRITESTART 17)
(defconstant READSTART 15)
(defconstant USER-DEFINED-TAG 16386)
(defconstant START-COMPUTATION-TAG 0)
(defconstant END-COMPUTATION-TAG 1)

;;-----------------------------------------------------------------
;; function next-matching-event
```

```
;;     starting at 'event', return the next event in the process
;;     history that satisfies the predicate matchfn or nil
;;     if no such event is found
;;-------------------------------------------------------------
(defun next-matching-event (event matchfn)
  (let ((last (event-last event)))
       (loop (when (funcall matchfn event) (return event))
             (when (equal last event) (return nil))
             (setq event (event-next event)))))

;;-------------------------------------------------------------
;; function proc-compute-time
;;     for the process at index 'procno' in execution 'history'
;;     return the length of the interval between the start-time
;;     of the first event in the history that satisfies the
;;     predicate 'startp' and the exit-time of the next event in
;;     the history that satisfies the predicate 'endp'
;;-------------------------------------------------------------
(defun proc-compute-time (history procno startp endp)
  (assert (s-history-p history))
  (let ((prochead (hist-process history procno)))
       (assert (s-event-p prochead))
       (let ((start (next-matching-event prochead startp)))
            (assert (s-event-p start))
            (let ((end (next-matching-event start endp)))
                 (assert (s-event-p end))
                 (- (event-exit end) (event-stime start))))))

;;-------------------------------------------------------------
;; function proc-interval-sum
;;     return the sum of closed intervals for the process
;;     that begins with 'event' where each closed interval begins
;;     with the start-time of an event that satisfies the
;;     predicate 'startp' and ends with the exit-time of an event
;;     that satisfies the predicate 'endp'.
;;-------------------------------------------------------------
(defun proc-interval-sum (event startp endp)
  (let ((current event)
        (time 0))
       (assert (s-event-p current))
       (loop
        (let ((start (next-matching-event current startp)))
             (if (s-event-p start)
                 (let ((end (next-matching-event start endp)))
                      (if (s-event-p end)
```

```
                        (block addtosum
                         (setq time
                           (+ time (- (event-exit end)
                                      (event-stime start))))
                         (setq current end))
                        (return time)))
                (return time))))))

;;-------------------------------------------------------------
;; function proc-interval-trace-se
;;     return a trace of closed intervals for the process
;;     that begins with 'event'. each closed interval begins
;;     with the start-time of an event that satisfies the
;;     predicate 'startp' and ends with the exit-time of an event
;;     that satisfies the predicate 'endp'.
;;-------------------------------------------------------------
(defun proc-interval-trace-se (event startp endp)
  (let ((current event)
        (trace '()))
       (assert (s-event-p current))
       (loop
        (let ((start (next-matching-event current startp)))
             (if (s-event-p start)
                 (let ((end (next-matching-event start endp)))
                      (if (s-event-p end)
                          (block addtotrace
                           (setq trace
                             (cons (- (event-exit end)
                                      (event-stime start))
                                   trace))
                           (setq current end))
                          (return (nreverse trace))))
                 (return (nreverse trace)))))))

;;-------------------------------------------------------------
;; function proc-interval-trace-ae
;;     return a trace of closed intervals for the process
;;     that begins with 'event'. each closed interval begins
;;     with the access-time of an event that satisfies the
;;     predicate 'startp' and ends with the exit-time of an event
;;     that satisfies the predicate 'endp'.
;;-------------------------------------------------------------
(defun proc-interval-trace-ae (event startp endp)
  (let ((current event)
        (trace '()))
```

```
        (assert (s-event-p current))
        (loop
         (let ((start (next-matching-event current startp)))
              (if (s-event-p start)
                  (let ((end (next-matching-event start endp)))
                       (if (s-event-p end)
                           (block addtotrace
                            (setq trace
                                  (cons (- (event-exit end)
                                           (event-access start))
                                        trace))
                            (setq current end))
                           (return (nreverse trace))))
                  (return (nreverse trace)))))))

;;*****************************************************************
;; function gauss-worker-procs-compute-time
;;     determine the COMPUTE time for each worker process in a
;;     Gaussian elimination program execution
;;     return a list of ordered pairs, one for each worker, of the
;;     form (processor compute-time)
;;*****************************************************************
(defun gauss-worker-procs-compute-time (history)
  (assert (s-history-p history))
  (let ((numprocs (hist-number-procs history)))
       (do ((procno 1 (+ procno 1))
            (timings '()
                     (cons (list
                            procno
                            (proc-compute-time history procno
                                     #'gauss-event-compute-start-p
                                     #'gauss-event-compute-end-p))
                           timings)))
           ((= procno numprocs)
            (nreverse timings)))))

(defun gauss-event-compute-start-p (event)
  (and (= (event-opcode event) USER-DEFINED-TAG)
       (= (event-value event) START-COMPUTATION-TAG)))

(defun gauss-event-compute-end-p (event)
  (and (= (event-opcode event) USER-DEFINED-TAG)
       (= (event-value event) END-COMPUTATION-TAG)))

;;*****************************************************************
```

```
;; function gauss-worker-procs-comm-time
;;     determine the COMMMUNICATION time for each worker process
;;     in a Gaussian elimination program execution
;;     return a list of ordered pairs, one for each worker, of the
;;     form (processor communication-time)
;;*****************************************************************
(defun gauss-worker-procs-comm-time (history)
  (assert (s-history-p history))
  (let ((numprocs (hist-number-procs history)))
       (do ((procno 1 (+ procno 1))
            (timings '()
             (cons (list
                    procno
                    (let ((event (next-matching-event
                                  (hist-process history procno)
                                  #'gauss-event-compute-start-p)))
                         (proc-interval-sum event
                                  #'gauss-event-comm-start-p
                                  #'gauss-event-comm-end-p)))
                   timings)))
           ((= procno numprocs)
            (nreverse timings)))))

;; predicate gauss-event-comm-start-p: true if event represents
;;     start of communication
(defun gauss-event-comm-start-p (event)
  (let ((opcode (event-opcode event)))
       (or
        (= opcode POLLREADSTART)
        (= opcode READSTART)
        (= opcode WRITESTART)
        (= opcode POLLWRITESTART))))

;; predicate gauss-event-comm-end-p: true if event represents
;;     end of communication
(defun gauss-event-comm-end-p (event)
  (let ((opcode (event-opcode event)))
       (or (= opcode READEND)
           (= opcode WRITEEND))))

;;*****************************************************************
;; function gauss-worker-procs-comm-trace-se
;;     collect a trace of COMMUNICATION in each round of the
;;     computation for each worker process in a Gaussian
;;     elimination program execution
```

```
;;     return a list of ordered pairs (one for each worker) of the
;;     form (processor (comm-time-round-1 ... comm-time-round-n))
;;*****************************************************************
(defun gauss-worker-procs-comm-trace-se (history)
  (assert (s-history-p history))
  (let ((numprocs (hist-number-procs history)))
       (do ((procno 1 (+ procno 1))
        (timings '()
             (cons (list
                procno
                (let ((event (next-matching-event
                      (hist-process history procno)
                      #'gauss-event-compute-start-p)))
                 (proc-interval-trace-se
                  event
                  #'gauss-event-comm-start-p
                  #'gauss-event-comm-end-p)))
               timings)))
       ((= procno numprocs)
        (nreverse timings)))))

;;*****************************************************************
;; function gauss-worker-procs-comm-trace-ae
;;     collect a trace of DATA TRANSFER TIME in each round of the
;;     computation for each worker process in a Gaussian
;;     elimination program execution
;;     return a list of ordered pairs (one for each worker) of the
;;     form (processor (xfer-time-round-1 ... xfer-time-round-n))
;;*****************************************************************
(defun gauss-worker-procs-comm-trace-ae (history)
  (assert (s-history-p history))
  (let ((numprocs (hist-number-procs history)))
       (do ((procno 1 (+ procno 1))
        (timings '()
             (cons (list
                procno
                (let ((event (next-matching-event
                      (hist-process history procno)
                      #'gauss-event-compute-start-p)))
                 (proc-interval-trace-ae
                  event
                  #'gauss-event-comm-start-p
                  #'gauss-event-comm-end-p)))
               timings)))
       ((= procno numprocs)
```

```
        (nreverse timings)))))

;;*****************************************************************
;; function gauss-worker-comm-ratio-se
;;      compute the RATIO of COMMUNICATION TIME / COMPUTATION TIME
;;      for the worker processes in a Gaussian elimination execution
;;      return a list of ordered pairs (one for each worker) of the
;;      form (processor ratio)
;;*****************************************************************
(defun gauss-worker-comm-ratio-se (comp-time comm-time)
  (mapcan #'(lambda (comp comm)
                    (list (/ (cadr comm) (float (cadr comp)))))
          comp-time comm-time))
```